

USENIX Association

Proceedings of the

Fourth Annual

Tcl/Tk Workshop

**July 10 - 13, 1996
Monterey, California**

Table of Contents

Fourth Annual Tcl/Tk Workshop

Thursday, July 11

INTERNET APPLICATIONS

Session Chair: Brent Welch, Sun Microsystems Laboratories, Inc.

Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application 1
Mark Roseman, University of Calgary

Agent Tcl: A Flexible and Secure Mobile-Agent System 9
Robert S. Gray, Dartmouth College

TclJava: Toward Portable Extensions 25
Scott Stanton and Ken Corey, Sun Microsystems Laboratories, Inc.

A Tk Netscape Plugin 31
Jacob Levy, Sun Microsystems Laboratories, Inc.

MISCELLANEOUS EXTENSIONS

Session Chair: David Young, The Information Refinery, Inc.

TclDG - A Tcl Extension for Dynamic Graphs 37
John Ellson and Stephen North, Lucent Technologies

Backtracking and Constraints in Tcl-BC 49
Dayton Clark and David M. Arnow, Brooklyn College

QuaSR: A Large-Scale Automated, Distributed Testing Environment 61
Steven Grady, G. S. Madhusudan and Marc Sugiyama, Sybase, Inc.

TclSolver: An Algebraic Constraint Manager for Tcl 69
Kevin B. Kenny, Manufacturing Technologies Laboratory, GE Corporate R&D Center

MISCELLANEOUS APPLICATIONS

Session Chair: Joseph A. Konstan, University of Minnesota

The NR Newsreader 75
Jonathan L. Herlocker, University of Minnesota

Tcl/Tk in the Development of User-Extensible Graphical User Interfaces..... 83
John M. Skinner and Robert M. Sweet, Brookhaven National Laboratory;
Richard S. LaBarca, Carnegie Mellon University

Visual Tcl: Building a Distributed MultiPersonality GUI Toolkit for Tcl 91
Mike Hopkirk, Santa Cruz Operation, Inc.

Friday, July 12

CORE Tcl/Tk TECHNOLOGIES

Session Chair: Michael J. McLennan, AT&T Bell Laboratories

An On-the-fly Bytecode Compiler for Tcl 103
Brian T. Lewis, Sun Microsystems Laboratories, Inc.

Tcl/Tk as an OpenDoc Scripting Part 115
Jim Ingham, AT&T Bell Laboratories

In Search of the Perfect Mega-widget	125
<i>Stephen A. Uhler, Sun Microsystems Laboratories, Inc.</i>	

INTEGRATING Tcl WITH OTHER SYSTEMS

Session Chair: Don Libes, NIST

SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++	129
<i>David M. Beazley, University of Utah</i>	

Automated Wrapping of a C++ Class Library into Tcl	141
<i>Ken Martin, GE Corporate R&D Center</i>	

Tksh: A Tcl Library for KornShell	149
<i>Jeffrey Korn, Princeton University</i>	

WORLD WIDE WEB PROGRAMMING

Session Chair: Ben Bederson, University of New Mexico

SurfIt! - A WWW Browser	161
<i>Steve Ball, Australian National University</i>	

Tcl/Tk HTML Tools	173
<i>Brent Welch and Steve Uhler, Sun Microsystems Laboratories, Inc.</i>	

Programming the Internet from the Server-Side with Tcl and Audience1™	183
<i>Adam Sah, Kevin Brown and Eric Brewer, University of California, Berkeley</i>	

Writing CGI scripts in Tcl	189
<i>Don Libes, NIST</i>	

SCIENTIFIC APPLICATIONS

Session Chair: Wayne Christopher, ICEM CFD Engineering

Lessons from the Neighborhood Viewer: Building Innovative Collaborative Applications in Tcl and Tk	203
<i>Alex Safonov, Douglas Perrin, Joseph Konstan, John Carlis and Robert Elde University of Minnesota</i>	

High Performance Graphic Display With Tcl/Tk	215
<i>Darren Spruce, European Synchrotron Radiation Facility</i>	

Hypertools in Image and Volume Visualization	221
<i>Pierre-Louis Bossart, Lawrence Livermore National Laboratory</i>	

A Clinical Neurophysiology Information System based on Tcl/Tk	231
<i>Martin B. Andrews and Richard C. Burgess, The Cleveland Clinic Foundation</i>	

WORKSHOP ORGANIZERS

Program Co-Chairs:

Mark Diekhans, Santa Cruz Operation, Inc.

Mark Roseman, University of Calgary

Program Committee

Ben Bederson, University of New Mexico

Wayne Christopher, ICEM CFD Engineering

Joseph A. Konstan, University of Minnesota

Don Libes, NIST

Michael J. McLennan, AT&T Bell Laboratories

Larry Rowe, University of California, Berkeley

Brent Welch, Sun Microsystems Laboratories, Inc.

David Young, The Information Refinery, Inc.

Will Wilbrink, Unisys Canada

Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application

Mark Roseman

*Dept. of Computer Science, University of Calgary
Calgary, Alta, Canada T2N 1N4*

Tel: +1-403-220-3532

E-mail: roseman@cpsc.ucalgary.ca

Abstract

This paper describes TeamRooms, a Tcl-based real time groupware application that provides “network places” for users to collaborate. TeamRooms is significantly more complex than previous groupware applications, providing not only generic tools such as shared whiteboards, but also custom groupware applets running within an OpenDoc-style embedded window. As well as describing TeamRooms itself, the paper relates the use of several Tcl programming techniques — meta-architectures, multiple interpreters, and embedded windows — that are used to manage the resulting complexity of the system.

Introduction

This paper describes a novel groupware application called TeamRooms, written using Tcl/Tk. Groupware systems provide a means for several users to work together, even though they may be separated by distance. TeamRooms approaches this problem by providing “network places” on the Internet, where users can gather to meet in real-time or can asynchronously leave information for each other. The metaphor is based on the physical team rooms used by many co-located work groups [4].

Previously, we had developed a number of applications in GroupKit, a Tcl/Tk extension or toolkit we had developed for building groupware [7]. TeamRooms was somewhat of a departure from these applications, demanding a different network architecture, more provisions for security and robustness, and needed to go cross platform. The user interface was to move from the relatively straightforward model of “one tool per window” to a model where several tools could exist in a single window, as found in compound document architectures such as OpenDoc or OLE [5].

In developing TeamRooms, we were faced with the following constraints: there was not enough time or resources to just completely rewrite everything, and it was important to keep the ease of building applications found in the original GroupKit. Even though the entire system was becoming much more complex, that added complexity had to be carefully managed and controlled.

This paper consists of two parts. The first part provides some background on real-time groupware and GroupKit, and then carries on to describe TeamRooms and its user interface. The discussion emphasizes the novel aspects of TeamRooms as a Tcl/Tk program: it is multi-user, multi-process, and an example of a highly-interactive Internet application. Its combination of several smaller Tcl programs with a compound document interface is also new.

The second part of the paper describes how TeamRooms was constructed, while still keeping our investment in existing GroupKit code and its straightforward API. The techniques used include meta-architectures, multiple interpreters, and embedded windows. Because some of these techniques may be applicable to managing the complexity in other Tcl/Tk programs, some problems that were faced along the way are also described.

About Groupware and GroupKit

Before delving into TeamRooms, some background is necessary. For those unfamiliar with the domain, this section first introduces real-time groupware applications. It then describes our GroupKit extension, and in particular the scope of applications which it has been possible to create with GroupKit.

Real-Time Groupware

Groupware is software that helps two or more people collaborate. It is a pretty general category that includes

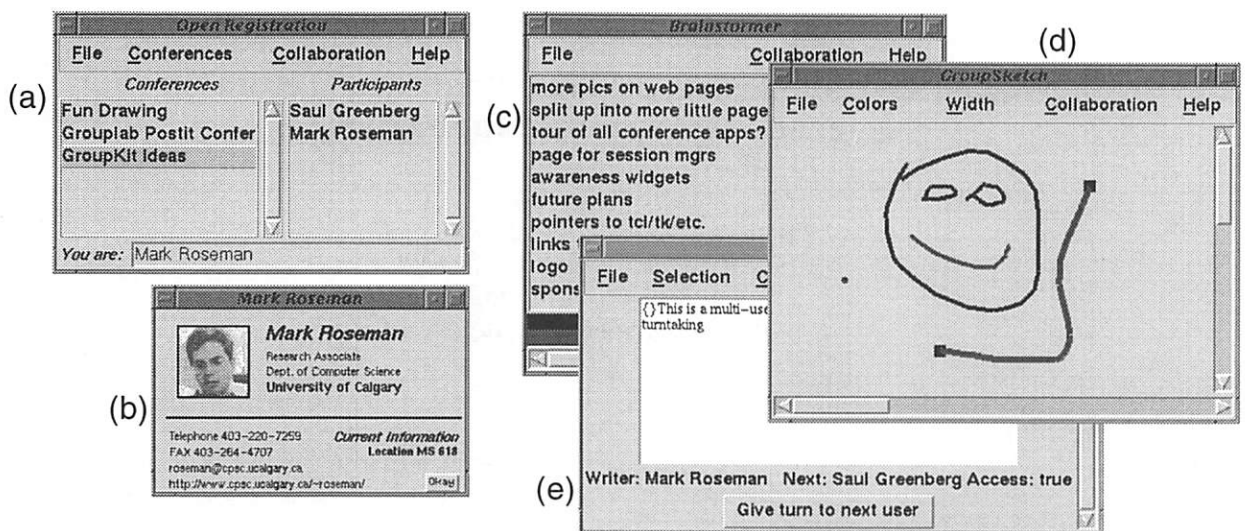


Figure 1. Some GroupKit applications, including (a) the open registration session manager, (b) information on a user, (c) a brainstorming tool, (d) a shared whiteboard, and (e) a shared text editor.

applications like e-mail or Usenet bulletin boards. Workflow and document management systems like Lotus Notes are the most commonly known groupware applications today.

Real-time groupware is groupware that lets people work together at the same time. A common example is the “talk” facility in Unix. Another example is a “shared whiteboard” program, that let people across a network draw together — any drawings marks made by one user on their computer are seen by all other users working on the shared drawing. Other examples are text editors that allow editing the same document at the same time (usually with some form of locking so users don’t conflict), brainstorming or voting tools for distributed meetings, card games, and so on.

GroupKit

Groupware can be both productive and fun to use. It is not, however, much fun to write. Even ignoring the considerable technical hurdles of network infrastructures and concurrency, there are many human factors issues that have to get worked out for anyone to be willing to use it. We developed GroupKit to make it easier for programmers to build real-time groupware applications. GroupKit is a toolkit or extension that relies on lower level support from Tcl, Tk, and Tcl-DP. Some of the facilities it provides to groupware developers are message passing, shared data structures, session management, and high-level multi-user interface widgets.

Figure 1 shows some typical applications constructed with GroupKit. The session manager is used to start each tool, which runs as its own process in its own window. When several users join a groupware session (for example, a shared whiteboard tool), each user’s process makes a socket connection to every other user’s process, which is known as a replicated architecture. Though GroupKit supports many different tools and even different session managers, the basic run-time architecture is always the same.

Just as Tcl/Tk have made single-user applications easy to build, GroupKit has made groupware applications easy to build. The toolkit’s learning curve is quick to climb, making it suitable when time is limited, such as for university class projects. It has been used at a large number of sites, and a number of substantial systems have been built using it. Its design has also made it easy to transform many existing single-user Tcl/Tk programs into groupware. The toolkit has also served well in supporting our own research interests of exploring groupware user interface issues. The combination of high-level programming constructs and ease of learning have made GroupKit one of the most popular groupware development platforms available today.

Challenges

Still, there were areas we wanted to explore where we were hindered, particularly as we started focusing more on interesting applications. Besides running on Unix,

we wanted to be able to deploy applications across platforms like Macintosh and Windows. Our fully replicated network architecture worked well in a world of stable workstations and networks, but can be problematic with unreliable machines and modem connections. Finally, we wanted to explore richer, more integrated environments, where several groupware tools were closely tied together, for example embedded inside other applications, documents or web pages.

TeamRooms

TeamRooms is our most ambitious groupware application to date. Unlike most of our tools which support isolated real-time meetings, the system provides a fully persistent environment for collaboration, whether in real-time or asynchronously. TeamRooms is modelled after physical team rooms, which provide a place for teams to meet, work, leave things for other team members, add comments and changes to shared documents, and so on. Our goal is to provide an electronic equivalent for teams whose members may be distributed. TeamRooms is a “network place” that hosts a team’s collaborations.

This metaphor is not new; Multi-User Dungeons (MUDs) also provide a persistent shared space, where people can meet in rooms containing various objects [3]. As with MUDs, TeamRooms uses a central server to hold information on rooms and their objects, and a separate client provides the user interface — but rather than a telnet client, TeamRooms has a full graphical interface (on Unix/X or Macintosh, with Windows under development). We wanted to move beyond the limited text-based interfaces of today’s MUDs, and provide “useful” fully interactive groupware applications as tools in the room. We stopped short of full audio/video support to keep network requirements reasonable, though an external system could be added.

User Interface

Figure 2 illustrates the user interface of the TeamRooms client, where the user (Mark) is in a room called “Mark Roseman’s Room” with two other users (Carl and Saul). Along the bottom of the screen are a text-based chat tool and different colored pens for drawing on the “walls” of the room (a shared whiteboard). User snapshots show who else is in the current room or on the server, and if a video camera is available, these pictures are periodically updated. Also shown are six different applets: a URL reference, an image, a concept map, a postit note, an outliner, and a tetriminoes game.

Applets

Each applet is embedded in its own frame, in a similar fashion as OpenDoc or OLE components. Users select new applets from the Tools menu, as well as delete, move and resize them. All changes are immediately visible to all users in the room. TeamRooms also allows users to retrieve earlier versions of applets, to compare changes over time.

Applets can be practically any groupware application, such as meeting tools (e.g. for brainstorming ideas or voting), shared document editors, drawing tools, or games. Some specific examples we built include:

PostIt. The ubiquitous yellow sticky note allows users to leave text messages in the room for other users, as reminders, or to comment on other room objects.

Outliner. A hierarchical outline tool lets users organize a set of notes or ideas. Users can add or delete ideas, drag existing ideas to rearrange them in the outline, and collapse or expand portions of the outline.

Image Tool. As a way to decorate rooms, we created the image applet, which displays a GIF image fetched from an HTTP server.

Games. The tetriminoes applet is one simple groupware game; others could include card games, chess or checkers, and so on.

URL References. To help tie in external information, this applet lets users leave pointers in the form of a URL for others. Clicking on the applet loads the requested URL into their web browser. Another applet uses Stephen Uhler’s infamous HTML parsing library to display a web page inside TeamRooms for discussion.

Applets differentiate TeamRooms from most groupware tools that provide only simple facilities such as chat rooms or shared whiteboards. Applets allow the environment to be customized to suit the team’s specific needs. Because we expected many users to want custom applets, we needed to make it easy to construct new ones, ideally as easy as constructing normal GroupKit applications.

Summary

TeamRooms provides a shared “network place” on the Internet where team members can collaborate, either in real-time or asynchronously. As a Tcl/Tk based Internet application, it is novel because of its multi-user, highly interactive nature, and its use of OpenDoc-style custom applets embedded inside the application.

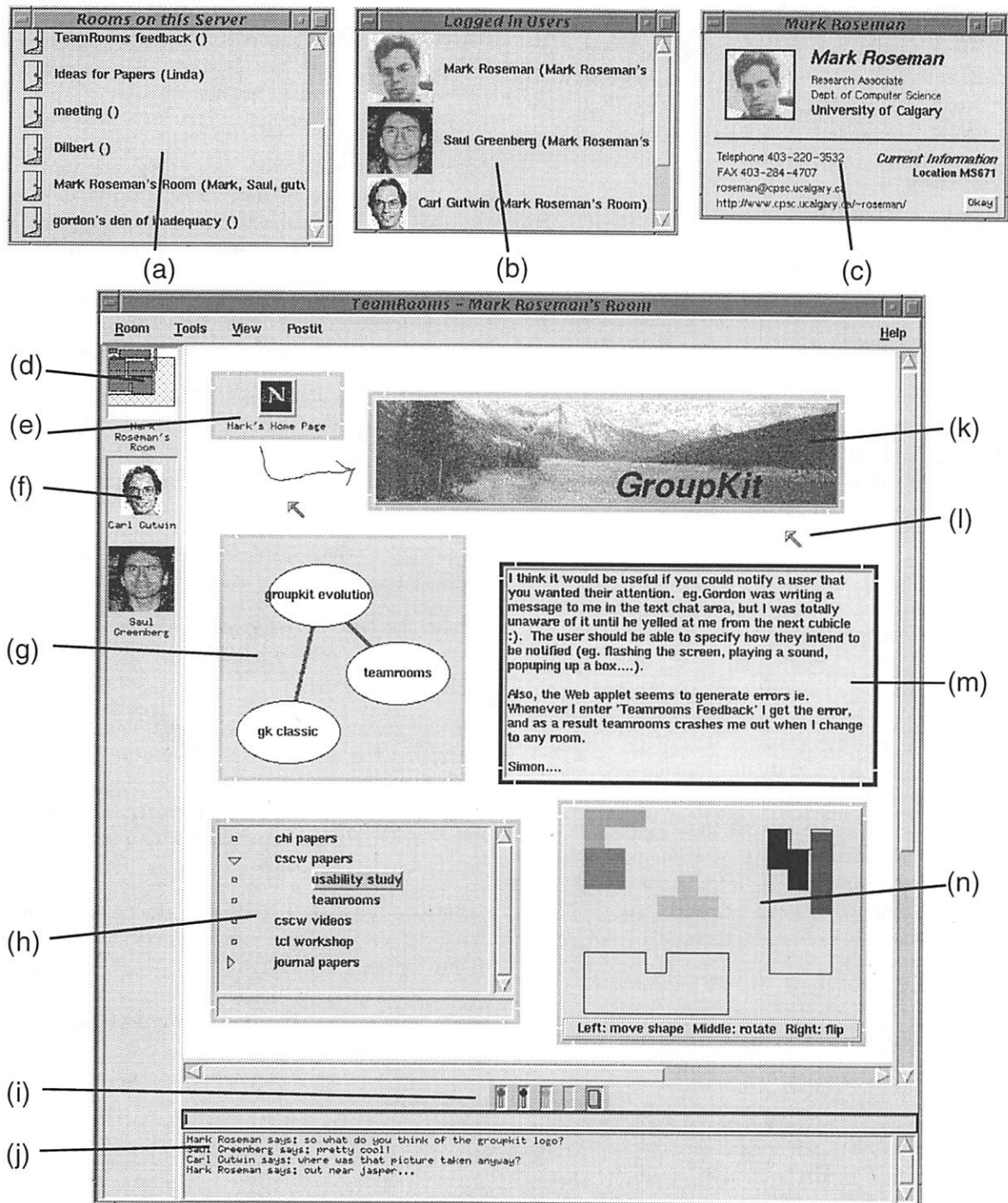


Figure 2. TeamRooms user interface, showing (a) available rooms; (b) connected users; (c) business card; (d) radar overview of room; (e) URL reference applet; (f) users in room; (g) concept map applet; (h) outliner applet; (i) whiteboard pens; (j) text chat area; (k) image applet; (l) telepointer; (m) postit applet; (n) tetrominoes applet.

Strategies for Managing Complexity

TeamRooms represents a rather significant challenge for a GroupKit application. Its architecture is centralized, not replicated; it requires user authentication; it demands a very robust, multi-versioned persistence facility; it needs to be multi-platform; and several groupware applications need to be embedded in the same toplevel window.

This section describes three techniques that were used to build TeamRooms while still leveraging the existing GroupKit code base and API where possible: meta-architectures, multiple interpreters, and embedded windows. After a description of these techniques, some of the particular issues that were encountered in building TeamRooms are addressed.

Meta-Architectures

Meta-architectures provide a way to change the underlying behavior of a software system while still retaining an existing interface or API. For example, we wanted to provide a centralized network architecture (new behavior), though still allowing developers to view the system as having direct connections to other processes for passing messages (a key component of the API).

In a meta-architecture, the user level API calls a small number of well-defined underlying primitives. The meta-architecture provides hooks to allow replacing those primitives. In GroupKit, we had primitives for opening, accepting and closing sockets, and passing messages. The existing primitives supporting a replicated architecture were replaced with new ones for a centralized architecture, and the user level routines continued to do the right thing. When it came time to add authentication (i.e. logins), we could again use the hooks to add the new behavior.

Building good meta-architectures comes down to good software design. It happens that highly dynamic languages like Tcl make them easy to implement. A more in-depth discussion on meta-architectures in Tcl is provided in [6].

Multiple Interpreters

The main problem for TeamRooms is dealing with all the different pieces: locating and navigating rooms, tools such as the shared whiteboard in the room itself, and then the numerous applets. Everything needs to be kept fairly separate and modular, while still being bundled together in the same application process.

Our first approach was to use an object system. A prototype of TeamRooms was built using [incr Tcl], where each applet was a mega-widget with groupware facilities added. While this worked, for this particular application it was not the ideal solution for two main reasons.

New Programming Model. Using an object system introduced a new programming model, where each groupware tool was an object. This added an extra level of complexity that we thought would be an obstacle to our target audience, most of whom are not experienced programmers or familiar with languages like C++. GroupKit's existing message passing paradigm was hard to resolve with objects, and imposing a particular structure on applications would impede the ability to adapt single-user applications.

Modularity Concerns. Surprisingly, modularity was also a concern. The burden was on the object's developer to ensure it did not use globals or otherwise interfere with other objects running in the application (despite interacting with its equivalent objects in other users' processes). This also had implications for security; though we were not immediately concerned with applets being downloaded over the network, the need to "trust" each object to interact nicely with the system seemed to preclude the possibility.

For the final version of TeamRooms, we abandoned objects and implemented the system with multiple interpreters, using the "stcl" extension that was added to the core in Tcl 7.5. Multiple interpreters allow us to view each piece as a completely separate groupware application that looks almost exactly like standard GroupKit code.

The TeamRooms client application consists of several GroupKit interpreters, as shown in Figure 3. The overall *application interpreter* is logically connected (via the central server) to all other users on the server, and deals with navigating between rooms, finding who is logged in to the server, and what rooms are available. When the user enters a room, a *room interpreter* is created to manage the overall room. Logically connected to all users in the room, this interpreter manages the chat tool, shared whiteboard, creates and resizes applets, etc. Finally, each applet runs in its own *applet interpreter*. Some advantages of this technique are noted below.

Standard Tcl Programs. The main advantage is that each component looks just like a standard "run-on-its-own" Tcl application (or in this case, GroupKit application). There are no extra constructs, and no special considerations to worry about. This perfectly addressed

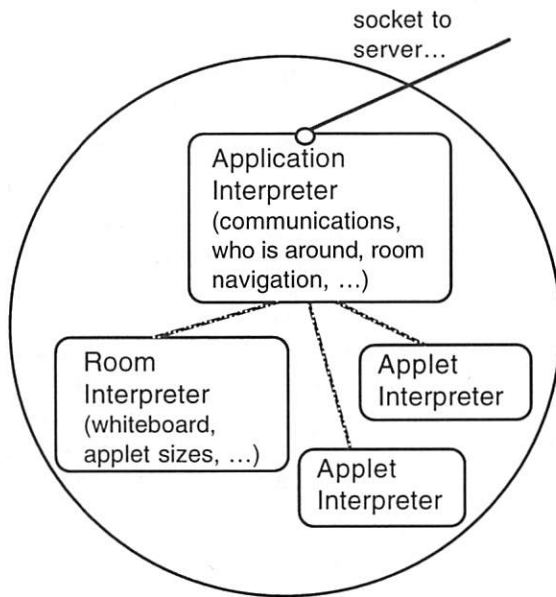


Figure 3. Structure of TeamRooms client, showing use of multiple interpreters.

our concerns with learning curve, and preserved our investment in existing applications.

Modularity and Security. Unlike with objects, programmers using multiple interpreters must explicitly go out of their way to access code out of the program's scope. This meant no accidental interference between applications. By providing a clear dividing line between pieces, it also makes it easy to replace pieces, such as the program for the room interpreter. Finally, this left open the possibility to enforce security restrictions on applets, using the model supported in Safe Tcl [2].

Shared Resources. Multiple interpreters were used to share resources across the entire application. For example, a single socket connection is shared between all interpreters. When an applet sends a message to its counterpart running in another user's client, the message is first routed to the application interpreter. It is then sent over the network to the server, which relays it to the application interpreter of the other user's client. From there, it is routed to the interpreter managing the specific applet. The actual mechanism was implemented by having the application interpreter set up an alias in the applet interpreter to intercept communications. Of course, the flexible routing scheme was specified using GroupKit's meta-architecture.

Embedded Windows

Though multiple interpreters give TeamRooms the needed lower level functionality, all those interpreters still had to be able to share the screen somehow.

Luckily, Steve Ball had already done most of the work for us in his SurfIt! web browser [1], which features Tcl applets running in their own subinterpreters and having access to Tk features. The basic approach is to carve off a piece of the Tk window hierarchy for the application interpreter, alias that to "." in the applet interpreter, and use aliases to redefine all Tk commands in the applet to run in the application interpreter, with appropriate changes to window names, etc.

We made several changes to this code. First, we allowed the window hierarchy of child interpreters to be rooted at an internal frame widget rather than only at a toplevel, so that interpreters could share the same toplevel window. We removed many of the security limitations enforced by SurfIt!, since at this point we wanted full access to Tk facilities. Finally, we moved several pieces of the code from Tcl into C to improve performance in critical areas.

The frame surrounding each applet is constructed as a standard Tk mega-widget (itself containing 20 small frames for the different pieces of the border), whose inside frame is the root of the applet's window hierarchy. We followed the practice found in the OpenDoc compound document framework [5] that the parent determines the layout of the child, so all resize decisions etc. are managed by the parent.

Issues

The previous section describes some of the techniques that we used in building TeamRooms. Because these may be applied to managing complexity in other Tcl/Tk applications, we now look at some of the obstacles that were faced in applying these techniques in TeamRooms, as well as the solutions we found.

Startup Time

The first difficulty had to do with startup time. Because each interpreter acts like its own application, starting up several different interpreters is like starting up several applications. While a two second initial application startup time may be reasonable, if it takes two seconds for every single applet to be created, the time it takes to enter a room in TeamRooms holding five or ten applets can seem like an eternity.

It took a lot of profiling (mostly using Tcl's "time" command) and subsequent performance tuning to get the time it took to create an applet interpreter and its frame down from about 2.5 seconds to a more reasonable .2 seconds. Some of the changes we made are described below. Note that most are common sense optimizations

that were just never an issue before, and that the typical “if its slow, recode it in C” would only address a small number of the problems in this case.

Do the minimum amount of work. Our subinterpreters took a lot of time initializing code they didn’t need. For example, we’d originally initialized Tcl-DP even though the applets used the application interpreter’s socket facilities (removing this saved about .15 seconds). We used to read one large Tcl configuration file, which included much information used only by other parts of TeamRooms; this was moved into a different file (saving about .2 seconds). Obviously, minimizing work is especially important if the work is done at the slower Tcl level, rather than C.

Avoid autoloading. While autoloading is a very convenient way to load Tcl source code, it is extremely slow! We explicitly sourced all scripts rather than relying on unknown handlers and auto-loading (total saving around .5 seconds, depending on the applet).

Identify special cases. One data structure we use is created and maintained mostly through Tcl code. When creating a new instance, the programmer may specify a number of different options, which requires a lot of slow Tcl code to parse. We identified a frequently-used special case and handled that separately. These types of optimizations saved about .2 seconds.

Use smarter Tcl constructs. We found many improvements here. Our best example is a construct like “lsearch [info commands] foo” rather than “info commands foo” which runs about forty times faster. While Tcl is great to “glue” primitives together, its worth checking the manual pages to see if your favourite Tcl command will do the work for you itself.

Embedded Window Issues

Most of the embedded window issues we faced were performance issues, not surprising given that the code to do the embedding was written in Tcl. In this case, profiling identified some special cases which were rewritten (e.g. there is no need to search through a command using an expensive regular expression search to find window pathnames if the character “.” doesn’t appear anywhere in the command), or some general routines which were used everywhere where it was worth it to rewrite just those routines in C.

Using mega-widgets was another issue. Both the mega-widget framework we used and the mega-widgets themselves were written in Tcl. Given the overhead of the window embedding code, both creating and using mega-widgets that run in the child interpreter was very

slow. Moving them into the parent interpreter (and making them available in the child interpreter with an alias, as is done with the built-in Tk widgets) improved that situation considerably (creating the mega-widget for the applet’s object frame took .5 seconds when run in the child interpreter, and just under .1 seconds when run in the parent interpreter).

There were a few other difficulties, such as not being able to access the “-variable” associated with some widgets in a subinterpreter (which was resolved by a set of variable traces). Deciding how images were shared between interpreters is also an issue (we let child interpreters have full access to the parent’s images, though this decision may have to be revised if we allow untrusted applets). These will need to be resolved as the “safe Tk” code is redone and integrated into the Tk core.

Interactions Between Interpreters

Interpreters need to communicate with each other to share facilities, such as sockets, information on users, and so on. The multiple interpreter package in Tcl uses a “parent/child” paradigm for interpreters, which we followed closely. Shared facilities were always supplied by the parent (the application interpreter) to the child (the room or applet interpreters), using interpreter aliases. This resulted in the application interpreter program needing extra code, while the code used in the room and applet stayed quite simple, which worked well for our need of simplifying applets.

Though it is possible to use hierarchical interpreters, after some brief experimentation we rejected them. With the applet interpreters being a child of the room interpreter (rather than the application interpreter), and even at one point with applets as children of other applets, things got out of control quickly. Speed was an issue (mainly in the interface code), and responsibilities were spread over many pieces. When possible, a shallow hierarchy of interpreters seems to be more effective.

Another decision we had to make was about menu sharing, so that applets could have access to the main menubar. We chose to add a single menu to the menubar for each applet (available via an alias), and the application interpreter packed and unpacked the menu as the focus changes. An alternative would be to clone the entire menubar for each applet.

Packaging

Because our audience is not only developers but also people who just want to use the system, we needed to package a binary that would not require users to compile

their own Tcl, Tk, GroupKit, etc. Existing solutions need some changes to work for applications using multiple interpreters. Typically these systems "compile" Tcl code into arrays of C strings, and load them via `Tcl_Eval()` at the start of the program. But multiple interpreters are not always created at the program's start, and interpreters may use different files.

The solution we used in TeamRooms was to use an existing package (Joe Touch's "tcl2array" package) to generate C arrays of the Tcl code. We then created a hash table containing pointers to these arrays, indexed by their original Tcl filename. We replaced Tcl's standard "source" command with a new version that first checks if the requested filename is in the table. If so, the code is read from the array, otherwise the file is read from disk.

Cross Platform Issues

While TeamRooms now runs on several flavours of Unix, Macintosh, and Windows, at the time of writing we have little to report in terms of cross platform issues that were difficult to resolve. Most difficulties have to do with missing native functionality (e.g. proper menus and dialog boxes), differences with fonts (which are important if we want identical views of the room across platforms), and so on. Other common cross-platform issues such as layout, naming conventions and so on have not been significant issues with TeamRooms. This is likely because the system relies on a very customized, direct-manipulation interface built using Tk's canvas widget, rather than using a more conventional forms based interface.

Conclusions

This paper has described TeamRooms, a Tcl/Tk groupware application built with our GroupKit toolkit. TeamRooms provides "network places" on the Internet for collaborators, who can interact with generic tools like shared whiteboards. They can also customize their electronic rooms by using applets, which are full groupware applications that run embedded in the room's window, OpenDoc style. TeamRooms is a good illustration of a highly interactive Tcl-based Internet environment.

To accomplish this while still keeping the application's complexity reasonable, TeamRooms relies heavily on a number of techniques. Meta-architectures provide the flexibility to support new run-time architectures. Multiple interpreters allow us to structure the system so that each component acts as its own self-contained application, without requiring extra knowledge about

the overall environment. Finally, embedded windows extend the power of multiple interpreters to Tk. Our experiences with these techniques should prove useful as other Tcl/Tk applications begin to use these newer features.

Acknowledgements

Thanks go to early users, in particular Saul Greenberg, Carl Gutwin, Gordon Paynter and Simon Gianoutsos, who braved early versions of the system, and offered many useful suggestions and improvements. Various bits of code have been borrowed from Steve Ball (applet embedding), Stephen Uhler (HTML library), and Shannon Jaeger (megawidget framework). The financial support provided by Intel Corporation and NSERC is gratefully appreciated.

More information about TeamRooms, including software availability, related projects, and publications, can be obtained on the World Wide Web at:

<http://www.cpsc.ucalgary.ca/projects/grouplab/teamrooms/>

References

1. Ball, S. *SurfIt! A WWW Browser*. In *Proc. of Tcl/Tk Workshop*. 1996..
2. Borenstein, N. *EMail with a Mind of its Own: The Safe-Tcl Language for Enabled Mail*. In *Proc. of ULPAAL*. 1994.
3. Curtis, P. and Nichols, D. *MUDs Grow Up: Social Virtual Reality in the Real World*. In *Proc. of the Third International Conference on Cyberspace*. May 1993.
4. Johansen, R., Sibbet, D., Benson, S., Martin, A., Mittman, R. & Saffo, P. *Leading Business Teams*. Addison-Wesley. 1991.
5. Orfali, R., Harkey, D. and Edwards, J. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons. 1996.
6. Roseman, M. *When is an object not an object?* In *Proc. of Tcl/Tk Workshop*. 1995.
7. Roseman, M. and Greenberg, S. *Building Real Time Groupware with GroupKit, a Groupware Toolkit*. *ACM TOCHI*. March 1996.

Agent Tcl: A flexible and secure mobile-agent system

Robert S. Gray*

Department of Computer Science

Dartmouth College

Hanover, New Hampshire 03755

robert.s.gray@dartmouth.edu

Abstract

An *information agent* manages all or a portion of a user's information space. The electronic resources in this space are often distributed across a network and can contain tremendous quantities of data. *Mobile agents* provide efficient access to such resources and are a powerful tool for implementing information agents. A mobile agent is an autonomous program that can migrate from machine to machine in a heterogeneous network. By migrating to the location of a resource, the agent can access the resource efficiently even if network conditions are poor or the resource has a low-level interface. Telescript is the best-known mobile-agent system. Telescript, however, requires the programmer to learn and work with a complex object-oriented language and a complex security model. Agent Tcl, on the other hand, is a simple, flexible, and secure system that is based on the Tcl scripting language and the Safe Tcl extension. In this paper we describe the architecture of Agent Tcl and its current implementation.

1 Introduction

An *information agent* is charged with the task of managing all or a portion of a user's information space. The electronic resources in this space are often distributed across a network and can contain tremendous quantities of data. *Mobile agents* allow efficient access to such resources and are a powerful tool for implementing information agents. A mobile agent is a program that can migrate *under its own control* from machine to machine in a heterogeneous network. In other words, an agent can suspend its execution at any point, transport its code

and state to another machine, and resume execution on the new machine. By migrating to the location of an electronic resource, an agent can access the resource locally and can eliminate the network transfer of all intermediate data. Thus the agent can access the resource efficiently even if network conditions are poor or the resource has a low-level interface. This efficiency, combined with the fact that an agent does not require a permanent connection with its home site, makes agents particularly attractive for mobile computing since roving devices often have a low-bandwidth, unreliable connection into the network. Mobile agents also ease the development, testing and deployment of distributed applications since they hide the communication channels but not the location of the computation [Whi94], they eliminate the need to detect and handle network failure except during migration, and they can dynamically distribute and redistribute themselves throughout the network. Mobile agents move the programmer away from the rigid client-server model to the more flexible peer-peer model in which programs communicate as peers and act as either clients or servers depending on their current needs [Coe94]. Finally, anecdotal evidence suggests that mobile agents are easier to understand than many other distributed-computing paradigms. Existing applications for mobile agents include electronic commerce, active documents and mail, information retrieval, workflow and process management, and network management [Whi94, Ous95]. Potential applications include most distributed applications, particularly those that must run on disconnected platforms or that must invoke multiple operations at each remote site [Whi94, Ous95].

The advantages and potential applications of mobile agents have led to a flurry of recent implementation work. Notable systems include Telescript from General Magic, Inc. [Whi94], Tacoma

*Supported by AFOSR contract F49620-93-1-0266 and ONR contract N00014-95-1-1204. A small section of this paper appeared in [Gra95].

from the University of Cornell [JvRS95], SodaBot from MIT [Coe94] and ARA from the University of Kaiserslautern [Pei96]. These systems suffer from a range of weaknesses. Telescript provides a complex, object-oriented language and a complex security model in which the *programmer* must carefully identify and disallow dangerous actions. Tacoma and SodaBot provide high-level scripting languages (Tcl and SodaBotL respectively) that are much easier to learn and use. In addition, Tacoma uses the Horus toolkit to provide significant fault tolerance. Tacoma, however, requires the programmer to explicitly capture state information before migration. Tacoma and SodaBot only partially address security issues — Tacoma via simple encryption and SodaBot via minimal user control over resource usage — and do not provide low-level communication mechanisms, forcing some communication to take place outside of the agent framework. Finally, although the scripting languages of Tacoma and SodaBot are sufficient for most tasks, the lack of a faster language makes them unsuitable for speed-critical applications. ARA strikes a balance between the Telescript and Tacoma extremes by providing multiple languages, a framework for incorporating additional languages, and low-level communication mechanisms. ARA, however, has not been released and does not address security issues.

Agent Tcl is a mobile-agent system that is under development at Dartmouth College [Gra95]. Agent Tcl, like ARA, attempts to strike a balance among existing systems. Agent Tcl uses the flexible scripting language Tcl as its main language but provides a framework for incorporating additional languages. Agent Tcl provides migration and communication primitives that do not require the programmer to explicitly capture state information and that hide the actual transport mechanisms *but* that are low-level enough to be used as building blocks for a range of protocols. Agent Tcl uses the simple Safe Tcl security model to protect a machine from a malicious agent and agents from each other. Agent Tcl allows agents to migrate from machine to machine *or* remain stationary and access resources from across the network, to create child agents to perform sub-tasks, and to communicate with other agents on the local and remote machines. It is intended as a general environment for distributed applications, both in the Tcl/Tk and larger computing communities, with the application developer selecting the migration, communication and creation strategy that is best for the given network, resources and task. Although Agent Tcl is far from complete, it is in ac-

tive use at several sites and has been used in several information-management applications. These applications demonstrate the convenience and efficiency of mobile agents.

Section 2 presents the Agent Tcl architecture. Section 3 describes the selection of Tcl as the “main” agent language and the current implementation. Section 4 discusses the security concerns associated with mobile code, our current Safe Tcl security mechanisms, and the security mechanisms that must be added to provide sufficient protection for both the machines and the agents. Finally, Sections 5 and 6 briefly examine several information-management applications and highlight future work.

2 Architecture

Agent Tcl has four main goals:

- Reduce migration to a single instruction like the Telescript *go* and allow this instruction to occur at arbitrary points. The instruction should not require the programmer to explicitly capture state information and should hide the actual transport mechanisms.
- Provide transparent communication among agents. The communication primitives should be flexible and low-level but should hide the actual transport mechanisms.
- Support multiple languages and transport mechanisms and allow the *straightforward* addition of a new language or transport mechanism.
- Provide effective security in the uncertain world of the Internet.

The architecture of Agent Tcl is shown in Figure 1. The architecture builds on the server model of Telescript [Whi94], the multiple languages of ARA [Pei96], and the transport mechanisms of two predecessor systems at Dartmouth [Har95, KK94]. The architecture has four levels. The lowest level is an API for the available transport mechanisms. The second level is a server that runs at each network site. The server performs the following tasks:

- *Status.* The server keeps track of the agents that are running on its machine and answers queries about their status.
- *Migration.* The server accepts each incoming agent, authenticates the identity of its owner,

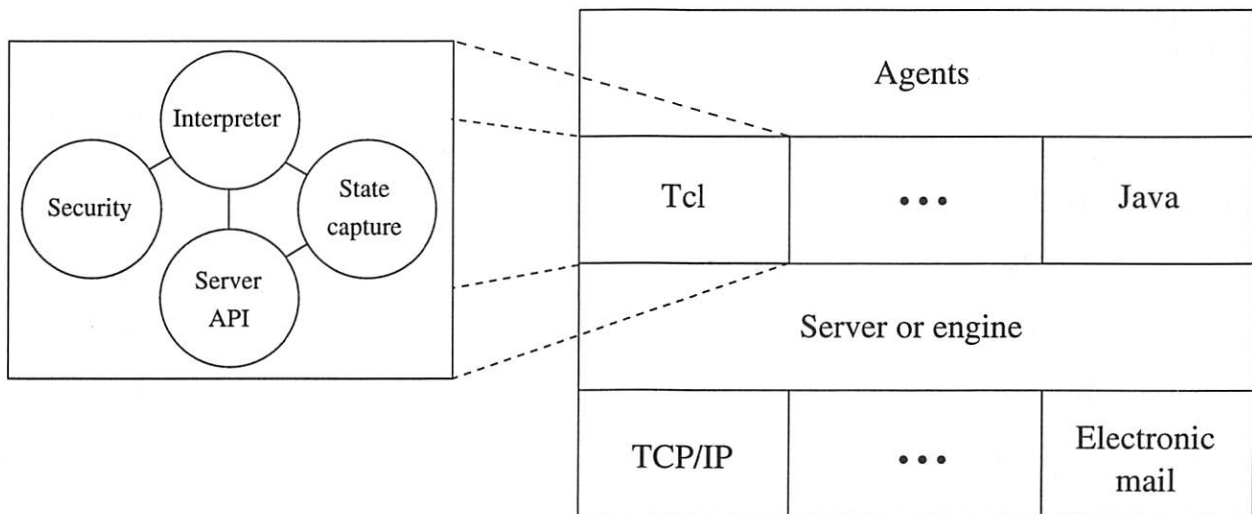


Figure 1: The architecture of Agent Tcl. The four levels consist of an API for the available transport mechanisms, a server that accepts incoming agents and mediates agent communication, an interpreter for each supported language, and the agents themselves.

and passes the authenticated agent to the appropriate interpreter. The server selects the best transport mechanism for each outgoing agent.

- *Communication.* The server provides a hierarchical namespace for agents and allows agents to send messages to each other within this namespace. The topmost division of the namespace is the network location of the agent. A message is an arbitrary sequence of bytes with no predefined syntax or semantics except for two types of distinguished messages. An *event* message provides asynchronous notification of an important occurrence while a *connection* message requests or rejects the establishment of a direct connection. A direct connection is a named message stream between agents and is more convenient and efficient than message passing (since the programmer can watch for messages on a particular stream and the server often can hand control of the stream to the interpreter). The server buffers incoming messages, selects the best transport mechanism for outgoing messages, and creates a named message stream once a connection request has been accepted.
- *Nonvolatile store.* The server provides access to a nonvolatile store so that agents can back up their internal state as desired. The server restores the agents from the nonvolatile store in the event of machine failure.

As in Tacoma all other services will be provided by *agents*. Such services include navigation, network sensing, group communication, fault tolerance, location-independent addressing, and access control. The most important service agents in our implemented prototype are *resource manager* agents which guard access to critical system resources such as the screen, network and disk. These resource managers are discussed in the security section.

The third level of the Agent Tcl architecture consists of one interpreter for each available language. We say *interpreter* since it is expected that most of the languages will be interpreted due to portability and security constraints (although “just-in-time” compilation is feasible for languages such as Java). Each interpreter has four components — the interpreter itself, a security module that prevents an agent from taking malicious action, a state module that captures and restores the internal state of an executing agent, and an API that interacts with the server to handle migration, communication, and checkpointing. Adding a new language consists of writing the security module, the state-capture module and a language-specific wrapper for the generic API. The security module does not determine access restrictions but instead ensures that an agent does not bypass the resource managers or violate the restrictions imposed by the resource managers. The state-capture module must provide two functions for use in the generic API. The first, *captureState*, takes an interpreter instance and constructs a machine-independent byte sequence that represents its inter-

nal state. The second, *restoreState*, takes the byte sequence and restores the internal state. The top level of the Agent Tcl architecture consists of the agents themselves.

3 Tcl and Agent Tcl

The architecture has not been completely implemented. The current implementation does not provide event messages or the nonvolatile store and has a single language (Tcl), a single transport mechanism (TCP/IP), and a *flat* rather than hierarchical namespace. It does provide migration, message passing and direct connections, and has sufficient security mechanisms to protect a machine from a malicious agent and to protect agents from each other. Incoming agents are authenticated using Pretty Good Privacy (PGP) [KPS95]; *resource manager* agents assign access restrictions based on this authentication; and Safe Tcl enforces these restrictions as the agent executes [BR]. Here we discuss the selection of Tcl as the main agent language and the details of the base system. We discuss security in the next section.

3.1 Tcl

Tcl is a high-level scripting language that was developed in 1987 and has enjoyed enormous popularity [Wel95]. Tcl has several advantages as a mobile-agent language. Tcl is easy to learn and use due to its elegant simplicity and an imperative style that is immediately familiar to any programmer. Tcl is interpreted so it is highly portable and easier to make secure. Tcl can be embedded in other applications, which allows these applications to implement *part* of their functionality with mobile Tcl agents. Finally, Tcl can be extended with user-defined commands, which makes it easy to tightly integrate agent functionality with the rest of the language and allows a resource to provide a package of Tcl commands that an agent uses to access the resource. A package of Tcl commands is more efficient than encapsulating the resource within an agent and is an attractive alternative in certain applications.

Tcl has several disadvantages. Tcl is a high-level, interpreted language so it is much slower than native machine code. In addition, Tcl provides no code modularization aside from procedures, which makes it difficult to write and debug large scripts. These disadvantages have not been a hindrance so far since mobile agents tend to involve high-level resource access wrapped with straightforward control

logic, a situation for which Tcl is uniquely suited. A mobile Tcl agent is usually short even if it performs a complex task, and is usually more than efficient enough when compared to resource and network latencies. In addition, several groups are working on structured-programming extensions to Tcl and on faster Tcl interpreters [Sah94]. Tcl is not suitable for every mobile-agent application, however, such as performing search operations against large, distributed collections of numerical data. For this reason, Agent Tcl includes a framework for incorporating additional languages. We are using this framework to add support for the new Java language [Sun94]. Java is much more structured than Tcl and has the potential to run at near-native speed through “just-in-time” compilation. We expect, however, that Tcl will continue to be the main agent language and that Java will be used only for speed-critical agents (or portions of agents).

The main disadvantage of Tcl is that it provides no facilities for capturing the *complete* internal state of an executing script. Such facilities are essential for providing transparent migration at arbitrary points. Adding these facilities to Tcl was straightforward but required the modification of the Tcl core. The basic problem is that the Tcl core evaluates a script by making *recursive* calls to `TclEval`. The handler for the `while` command, for example, recursively calls `TclEval` in order to evaluate the body of the loop. Thus a portion of the script’s state is on the C runtime stack and is not easily accessible. Our solution adds an explicit stack to the Tcl core. We split the command handlers into one or more *subhandlers* where there is one subhandler for each code section before or after a call to `TclEval`. Each call to `TclEval` is replaced with a push onto the stack. `TclEval` iterates until the stack is empty and always calls the current subhandler for the command at the top of the stack. The subhandlers are responsible for specifying when the command has finished and should be popped. Figure 2 illustrates this process for the `while` command.

It is important to note that the modified Tcl core is compatible with the standard Tcl core. A command procedure that makes a recursive call to `TclEval` will work correctly on top of the modified core; it will just be impossible to capture the script’s complete state when that command procedure is on the invocation stack. This means that existing Tcl extensions will work without modification (as long as the extension does not use the `tclInt.h` header file). An extension has to be modified only if the developer wants an agent to be able to carry the

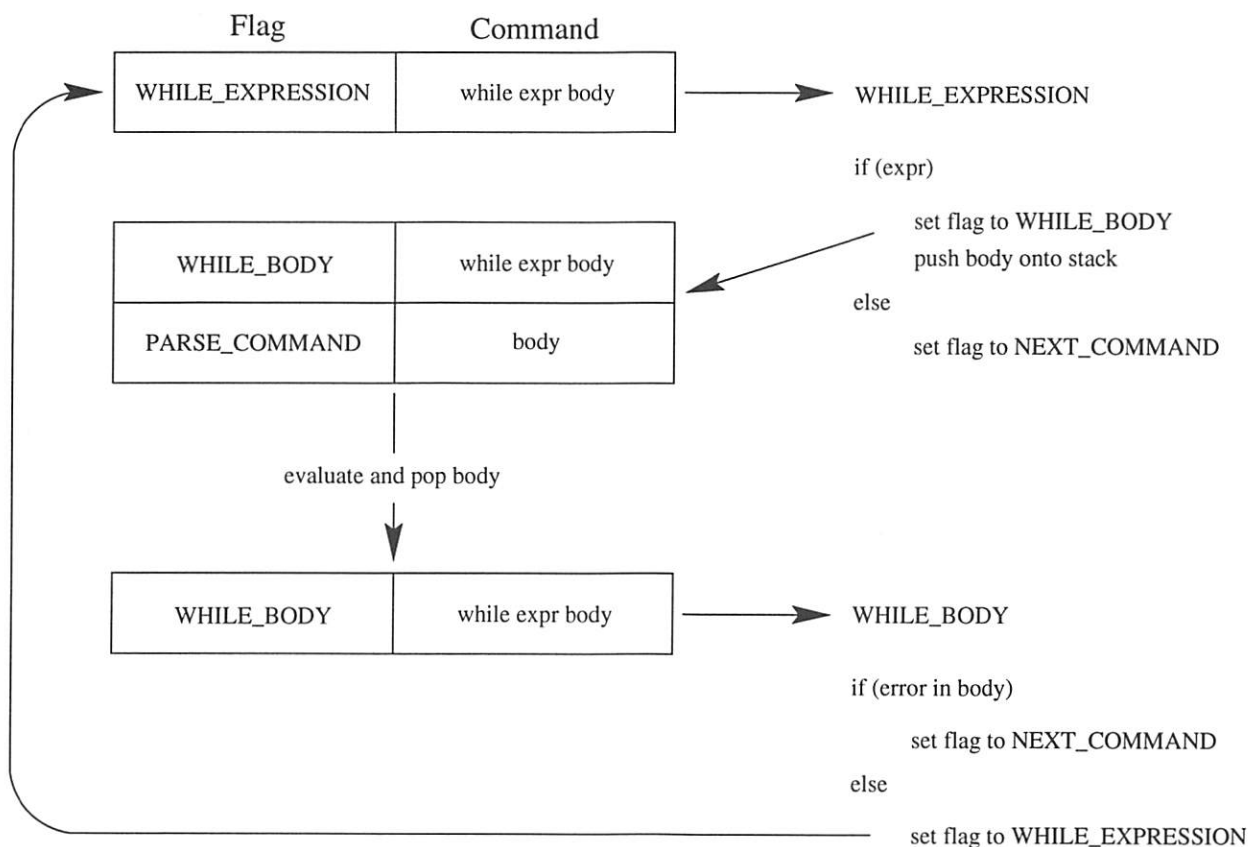


Figure 2: An example of how the stack works. The command stack is on the left and the two subhandlers for the `while` command are on the right. A subhandler sets the **NEXT_COMMAND** flag when the `while` command has finished and should be popped.

extension's state from machine to machine. In this case, the developer must make the same changes as for the `while` command and must provide callback routines for state capture and restoration.

The explicit stack is simpler and more flexible than the ARA solution, in which the C runtime stack must be captured in a portable way and the Tcl interpreter on the destination machine must contain the same set of C functions [Pei96]. On the other hand, the explicit stack is less efficient. Our modified Tcl core runs Tcl programs approximately 20 percent slower than the standard Tcl core, whereas ARA's modified Tcl core imposes little additional overhead. It appears that this performance penalty can be reduced significantly with additional optimization, however, and it would also be possible to include both the standard and modified Tcl cores within the same interpreter so that an agent could run on top of the standard, faster core if it did not want to migrate in mid-execution.

Once the explicit stack was available, it became

trivial to write procedures that save and restore the internal state of a Tcl script. These two procedures are the heart of the state-capture module for the Tcl interpreter. They capture and restore the stack, the procedure call frames, and all defined variables and procedures. Such things as open files and linked variables are currently ignored.

The advantages of Tcl are strong and the disadvantages are either easily overcome or do not affect most agents. Thus Tcl was chosen as the main language for the Agent Tcl system. The same advantages have led to the use of Tcl in other mobile-agent systems such as Tacoma [JvRS95] and ARA [Pei96].

3.2 Agent Tcl

The current implementation of Agent Tcl has two components. The first component is the server that runs at each network site. The server accepts, authenticates and starts incoming agents, buffers incoming messages, provides the flat namespace, and answers queries about the status of the agents that

are running on its machine. The server is implemented as two cooperating processes. One process watches the network while the other maintains a table of running agents.

The second component consists of a modified version of Tcl 7.5 and a Tcl extension. The modified version of Tcl 7.5 provides the explicit stack and the state-capture routines. The extension provides the commands that an agent uses to migrate, communicate, and create child agents. The most important commands are `agent_begin`, `agent_submit`, `agent_jump`, `agent_send`, `agent_receive`, `agent_meet`, `agent_accept`, and `agent_end`. Internally each command uses the server API to contact an agent server, transfer an agent, message or request, and wait for a response. The main difference between the current implementation and the proposed architecture is that when migrating, creating a child agent, or sending a message, the current implementation bypasses the local server and interacts directly with the destination server over TCP/IP. This approach was adopted to simplify the initial implementation and will change as additional transport mechanisms are added.

An agent is simply a Tcl script that runs on top of the modified version of Tcl 7.5. The agent uses the `agent_begin` command to register with a server and obtain a name in the flat namespace. A name currently consists of the IP address of the server, a unique integer, and an optional symbolic name that the agent specifies later with the `agent_name` command. The `agent_submit` command is used to create a child agent on a particular machine. The `agent_submit` command accepts a Tcl script, encrypts and digitally signs the script, and sends the script to the destination server. The server authenticates this agent, selects a name for the agent, and starts a Tcl interpreter in which to execute the agent. If the agent wants a symbolic name as well as a unique, integer identifier, it can call `agent_name` once it starts executing. The `agent_jump` command migrates an agent to a particular machine. The `agent_jump` command captures the internal state of the agent, encrypts and digitally signs the state image, and sends the state image to the destination server. The server authenticates this agent, selects a new name for the agent, and starts a Tcl interpreter. The Tcl interpreter restores the state image and resumes agent execution at the statement immediately after the `agent_jump`.

The `agent_send` and `agent_receive` commands are used to send and receive messages. The `agent_meet` and `agent_accept` commands are used

to establish a direct connection between agents. A direct connection is a named message stream. Direct connections are not required for communication but are more efficient and convenient as noted above. The source agent uses `agent_meet` to send a connection request to the destination agent. The destination agent uses `agent_accept` to receive the connection request and send either an acceptance or rejection. An acceptance includes a TCP/IP port number to which the source agent connects. The protocol works even if both agents use `agent_meet`. The agent with the lower IP address and integer identifier selects the port and the other agent connects to that port. A flexible RPC mechanism has been built on top of the direct connection mechanism [NCK96]. The server will take on more of the responsibility for establishing a direct connection as additional transport mechanisms are added.

Agent Tcl also includes a (slightly) modified version of Tk 4.1 so that an agent can present a graphical interface and interact with the user of its current machine. Event handlers can be associated with incoming messages and with direct connections.

4 Security in Agent Tcl

A mobile agent is a *program* that moves from machine to machine and executes on each. Neither the agent nor the machines are necessarily trustworthy. The agent might try to harm the machine or access privileged resources. The machines might try to pull sensitive information out of the agent or change the behavior of the agent by removing, modifying or adding to its data and code. Whether the agents and machines are actively malicious or programmed poorly, the end effect is the same. A mobile-agent system must provide security mechanisms that detect and prevent malicious actions. Without strong security mechanisms, a mobile-agent system will justifiably never be accepted and used. Security is perhaps the most critical issue in a mobile-agent system and can be divided into four interrelated problems:

- *Protect the machine.* The machine should be able to authenticate the agent's owner, assign access permissions based on this authentication, and prevent any violation of the access permissions.
- *Protect other agents.* An agent should not be able to interfere with another agent or steal that agent's resources. This problem can be viewed as a subproblem of protecting the machine, since as long as an agent cannot subvert

the agent-communication mechanisms and cannot consume or hold excessive system resources, it will be unable to affect another agent unless that agent chooses to communicate with it.

- *Protect the agent.* A machine should not be able to tamper with an agent or pull sensitive information out of the agent without the agent's cooperation. Clearly it is impossible to prevent a machine from doing whatever it wants with an agent that is currently executing on that machine. Instead we must detect tampering as soon as the agent migrates from a malicious machine back to an honest machine and terminate or fix the agent if tampering has occurred. In addition we must ensure (1) that the sensitive information never passes through an untrusted machine in an unencrypted form, (2) that the information is meaningless without cooperation from a trusted site, or (3) that theft of the information is not catastrophic and can be detected via an audit trail.
- *Protect a group of machines.* An agent might consume excessive resources in the network as a whole even if it consumes few resources at each machine. Obvious examples are an agent that roams through the network forever or an agent that creates two child agents, each of which creates two child agents in turn, and so on. An agent and its children should eventually be unable to obtain any resources anywhere and terminated.

All of these problems have been considered in the mobile-agent literature [LO95, CGH⁺95, TV96] although only the first two have seen significant implementation work. These same two problems are addressed in the current implementation of Agent Tcl using PGP [KPS95] and Safe Tcl [BR]. First we present the current implementation and then potential solutions for the remaining two security problems.

4.1 Authentication

Authentication in Agent Tcl is based on PGP (Pretty Good Privacy) which is in widespread use despite controversies over export restrictions and patents [KPS95]. PGP encrypts a file or mail message using the IDEA private-key algorithm and a randomly chosen private key, encrypts the private key using the RSA public-key algorithm and the recipient's public key, and then sends the encrypted key and file to the recipient. PGP optionally adds

a digital signature by computing an MD-5 cryptographic hash of the file or mail message and encrypting the hash value with the sender's private key. Although PGP is oriented towards interactive use, it can be used in an agent system with small modifications. In the current implementation we run PGP as a separate process, save the data to be encrypted into a file, ask the PGP process to encrypt the file, and then transfer the file to the destination server. This structure is much less efficient than tightly integrating PGP with the rest of the system, but is simpler and more flexible, especially since it becomes trivial to create an Agent Tcl distribution that does *not* include PGP or that uses different encryption software [Way95].

When an agent registers with a server using the `agent_begin` command, the registration request is digitally signed using the owner's private key, encrypted using the server's public key, and sent to the server. The server makes sure that the agent's owner is allowed to register on its machine and records the authenticated identity of the agent's owner. Then the IDEA private key is used as a session key for all further communication between the agent and its newly registered server. The session key is needed to prevent a malicious program from contacting the server and masquerading as an existing agent. When the agent and its registered server are on the same machine (which is the predominant case), we do not actually encrypt with the session key since there is no possibility of message interception; instead the session key is simply included in the message and compared against the server's recorded session key. A sequence number is included in the messages to prevent replay attacks.

When an agent migrates using the `agent_jump` command, it is digitally signed with the current server's private key and encrypted with the recipient server's public key. As in Telescript, we digitally sign using the *server's* private key since the owner's private key is unavailable once the agent leaves its home machine [TV96]. This approach requires the servers to have a high degree of trust in each other, so we will eventually adopt the Itinerant Agent solution [CGH⁺95], in which as much of the agent as possible is encrypted with the owner's private key on creation and remains encrypted throughout the agent's lifetime. The identity of the agent's owner is included in the migration message. The recipient server chooses whether to believe that identity based on its trust in the sending server. If the server accepts the agent, it records the apparent identity of the agent's owner, the authenticated identity of

the sending server, and its degree of confidence that the owner's identity is valid. A session key is used for all further agent-server communication as in the `agent_begin` case. The same steps occur when a child agent is created with the `agent_submit` command except that a Tcl script is encrypted rather than a Tcl state image. The same steps also occur when an agent sends a message to an agent on another machine. In the case of a direct connection, the IDEA private key from the acceptance message becomes the session key for the direct connection. A sequence number associated with the direct connection prevents replay attacks.

There are two weaknesses with the current implementation. First, there is no automatic distribution mechanism for the PGP public keys. Each server must already *know* all possible public keys so that it can authenticate incoming agents. An automatic distribution mechanism must be added when we start to use Agent Tcl in wide-area networks. Second, the system is vulnerable to replay attacks in which an attacker replays a migrating agent or any message sent from one agent to another (outside of a direct connection). An obvious solution is for each server to have a distinct sequence number for all servers with which it is in contact.

4.2 Authorization and enforcement

Once the identity of an agent's owner has been determined, the system must impose access restrictions on the agent (*authorization*) and ensure that the agent does not violate these restrictions (*enforcement*). In other words, the system must guard access to all available resources. We divide resources into two types. *Indirect* resources can only be accessed through another agent. *Builtin* resources are directly accessible through language primitives for reasons of efficiency or convenience or simply by definition. Builtin Tcl/Tk resources include the screen, the file system, wall-clock time and CPU time.

For indirect resources, the agent that controls the resource enforces the relevant access restrictions. For each message from another agent, the local server attaches to the message a 5-tuple that contains the apparent identity of the agent's owner, the apparent identity of the sending server, a flag that indicates whether the owner could be authenticated, a flag that indicates whether the sending server could be authenticated, and a numerical confidence level that represents how much trust the local server places in the sending server. The agent uses this 5-tuple along with its own internal access lists

to respond appropriately to the incoming message.

For builtin resources, security is maintained using Safe Tcl and a set of *resource manager* agents. Safe Tcl is a Tcl extension that is designed to allow the safe execution of untrusted Tcl scripts [BR]. Safe Tcl provides two interpreters. One interpreter is a "trusted" interpreter that has access to the standard Tcl/Tk commands. The other interpreter is an "untrusted" interpreter from which all dangerous commands have been removed. The untrusted script executes in the untrusted interpreter. Dangerous commands include obvious things such as opening or writing to a file, creating a network connection, and creating a toplevel window. Dangerous commands also include more subtle things such as ringing the bell, raising and lowering a window, and maximizing a window so that it covers the entire screen. Some of the subtle security risks do not actually involve damage to the machine or access to privileged information but instead involve serious annoyance for the machine's owner. The idea with this type of security risk is to restrict the number of times per second that the agent can *initiate* the event itself, or to restrict the agent to its own window in which it can do whatever it wants but whose size and position it can not affect.

Although the dangerous commands have been removed from the untrusted interpreter, we do not want to deny all access to the resources associated with these commands. Thus, instead of removing a dangerous command entirely, Safe Tcl can replace the command with a *link* to a command in the trusted interpreter. This trusted command either severely restricts the functionality of the original command or examines the command arguments and the identity of the script's owner to determine if the command should be allowed.

Agent Tcl uses the generalization of Safe Tcl that appears in the Tcl 7.5 core [LO95]. Agent Tcl creates a trusted and untrusted interpreter for each incoming agent. The agent executes in the untrusted interpreter. All dangerous commands have been removed from the untrusted interpreter and replaced with links to secure versions in the trusted interpreter. These secure versions check a set of access lists to see if the command is allowed. In the current implementation there is an access list for wall-clock and CPU time, the screen, the network, the file system, and external programs. Each access list is a set of (*name*, *quantity*) pairs where *name* specifies the name of the required resource and *quantity* specifies the number of instances of that resource (if applicable). The *screen* access list, for example, might con-

tain the pair (*toplevel*, 5), which indicates that the agent can have no more than five *toplevel* windows. The *program* access list might contain the pair (*ls*, ()) which indicates that the agent is allowed to execute the Unix *ls* program. Initially the access lists are empty except that the agent is given a minimal amount of wall-clock and CPU time (our modified Tcl interpreter aborts a script if the script exceeds the time limits). To obtain additional time or to obtain access to other builtin resources, the agent must explicitly or implicitly ask a *resource manager* agent for permission. There are five resource managers in the current system. These managers correspond to the five access lists and control access to wall-clock and CPU time, the screen, the network, the file system, and external programs.

An agent uses the **require** command to explicitly ask a resource manager for access. The **require** command takes the symbolic name of the resource manager — e.g., *screen* — and a list of (*name*, *quantity*) pairs which specify the desired access permissions — e.g., (*toplevel*, 5), (*screen_area*, 30 percent). The **require** command is actually just a link to a procedure in the trusted interpreter. This procedure contacts the appropriate resource manager and passes the list of access requests to the resource manager. The procedure waits for the response and then adds each access request to the internal access lists, indicating for each whether the request was granted or denied.

To implicitly ask a resource manager for access, an agent simply calls a command that uses the resource. For example, if the agent issues the command **exec ls**, the **exec** procedure in the trusted interpreter checks the *program* access list. If permission to execute *ls* has already been granted, the command proceeds. If permission to execute *ls* has already been denied, the command aborts with a security error. Otherwise the command contacts the *program* resource manager and either proceeds or aborts depending on the manager's response. Although these implicit access restrictions are convenient, the agent should use the **require** command whenever possible so that it can determine whether a required resource is available before it tries to use the resource.

Our implementation does not yet provide a safe version of all dangerous commands. For example, an agent that arrives from another machine can not use the **source** and **send** commands (the **send** command will probably never be available since it is difficult to make secure and agents should communicate within the agent framework anyways). In addition, the "annoyance" security threats have not been eliminated.

Rather than restricting the use of all associated commands, we plan to provide each agent with a virtual screen in which it can do whatever it wants but that only the user can move and resize. Although the annoyance threats remain, Agent Tcl currently protects the machine well using the simple kernel-user model of Safe Tcl. No direct access to system resources is possible. There is no way for an agent to subvert the *resource-manager* system since there is no way for the agent to modify the access lists contained in the trusted interpreter; it is possible for the agent to contact a resource manager directly, but this accomplishes nothing since the response (1) will correctly grant or deny access and (2) even so will not be added to the access lists. In our case, Safe Tcl is the mechanism for enforcing the policy provided by the resource managers. When Java is added to the system, the existing Java security mechanisms will be used to enforce the same policy provided by the same resource managers.

In addition to the resource managers, Agent Tcl includes a *console* agent, which is used primarily on machines that have a specific owner. The *console* agent has two purposes. First, it tracks all of the agents that are running on the machine, and allows the machine's owner to deny entry to incoming agents and to terminate running agents. Second, it provides a pathway through which a resource manager can ask the owner whether an agent should be able to perform a particular action. The owner will eventually be able to specify exactly those situations in which she should be asked.

4.3 Future security work

There are three areas of future work. First, we plan to add a hierarchical system of resource managers. This will become particularly important as we move towards the Telescript model in which there are multiple virtual *places* per machine [Whi94]. Each place might have its own security policy while the machine has an overall security policy. Second, we must protect an agent from malicious machines. Here we are exploring the suggestions from [CGH⁺95] in which an agent is divided into components and each component is encrypted and signed separately for all or part of the journey. This scheme allows immediate detection of blatant tampering, such as dropping part of the agent or inserting an entirely new procedure, and prevents blatant theft of sensitive data. In addition we plan to record an audit trail that can be analyzed to determine the point at which a failed agent might have been modified inappropri-

ately. For more subtle modification threats, such as modifying a piece of data that changes on every machine and thus must be unencrypted, solutions are less clear and may be impossible. Third, we must protect a group of machines from a malicious agent. Here we are looking at a *currency-based* resource-allocation scheme in which an agent's owner gives the agent a finite currency supply from her own finite currency supply. The currency does not have to be tied to legal currency, but it should be impossible to spend a currency unit more than once. The agent must spend currency in order to access resources and must divide its own currency among its children. The agent and its children will eventually run out of currency and terminate. Such currency schemes already exist in the context of electronic commerce [Way95].

5 Applications

Figure 3 shows the "who" agent which illustrates the agent commands. The agent's task is to determine which users are logged onto a set of machines. The agent uses `agent_submit` to create a child agent. The child agent jumps from machine to machine using `agent_jump` and executes the Unix `who` command on each machine. The child then sends the list back to its parent with `agent_send`. The parent has been waiting for the list with `agent_receive` and displays the list to the user.

Although its task is simple and can be accomplished easily without a mobile agent, the "who" agent illustrates the general form of any agent that migrates through a sequence of machines. Existing Agent Tcl agents that fall into this category are a workflow agent that carries an electronic form from user to user [CGN96] and a medical agent that retrieves distributed medical records based on certain criteria [Wu95]. The workflow agent must migrate sequentially since the users need to fill out the sections of the form in order. The medical-retrieval agent chooses to migrate sequentially since the agent can discard potential candidates as it travels through the distinct databases; spawning one child agent per remote database or interacting with the databases using the traditional client/server approach increases the total network traffic even when only a single operation is being performed against each database.

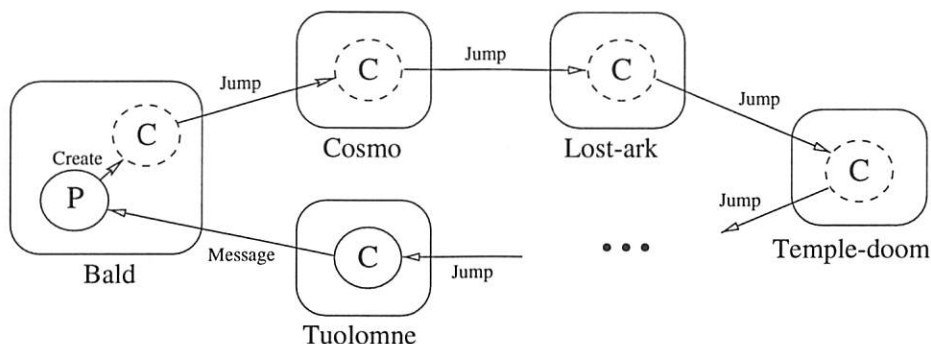
Like the "who" agent, the workflow and medical agents do not require continuous contact with the home machine and will continue their task even if the home machine becomes temporarily disconnected.

In addition, the workflow and medical agents are extremely easy to implement within the agent framework. The code is written as if every resource is local to the agent; the only difference is that the `agent_jump` command is used to move the agent from one machine to the next. The `agent_jump` command is not strictly necessary since we could continually resubmit a Tcl procedure that was parameterized according to the current status of the task; the procedure would use the parameters to determine what it needed to do on the current machine [JvRS95]. Such an approach, however, requires that the programmer explicitly collect the necessary state information. In the "who" agent, this state information is nothing more than an index into the machine list, but more and more state information is required as the agent becomes more complex. The `agent_jump` command is convenient since it automatically captures this state. The `agent_jump` command does impose a moderate execution overhead on the Tcl interpreter; this overhead can be made much smaller, however, and can even be reduced to near zero with the ARA solution [Pei96].

Another example is our "alert" agent that monitors a specified set of remote resources and notifies its owner of any change in resource status. Figure 4 shows an "alert" agent that monitors a set of files and notifies the user if the status of a file changes significantly (monitored characteristics include the Unix *rx* bits and the file size). The agent creates one child agent for each remote filesystem using `agent_submit`. Each child agent monitors one or more files in its filesystem and sends a message to the parent when the status of a file changes significantly. The parent then contacts the owner's "mail" agent to send an email message.

Since the child agents know which status changes are "significant", only the status changes that the user actually wants to see are transmitted across the network. Without mobile agents, either the remote machine would have to send back a notification of every change (which the application would filter on the home machine) or the appropriate monitoring routines would have to be pre-installed on the remote machine, limiting the application to the changes that the remote administrator considers significant. With mobile agents, the application can monitor for status changes according to any desired criteria while minimizing the ongoing network traffic.

A hybrid of the two examples is our text-retrieval agent that searches distributed collections of text documents. This agent is designed to be launched from a mobile device. It first obtains the query from



```

-----
# procedure WHO is the child agent that does the jumping
proc who machines {
    global agent
    set list ""

    # jump from machine to machine and execute the Unix who command on each machine
    foreach m $machines {
        if {catch "agent_jump" $m} {
            append list "$m:\n unable to JUMP to this machine"
        } else {
            set users [exec who]
            append list "$agent(local-server):\n$users\n\n"
        }
    }

    agent_send $agent(root) $list
    exit
}

set machines "bald cosmo lost-ark temple-doom moose muir tenaya tioga tuolomne"

# get a name from the server
agent_begin

# submit the child agent that jumps
agent_submit $agent(local-ip) -vars machines -procs who -script {who $machines}

# wait for and output the list of users
agent_receive code string -blocking
puts $string

# agent is done
agent_end
-----

```

```

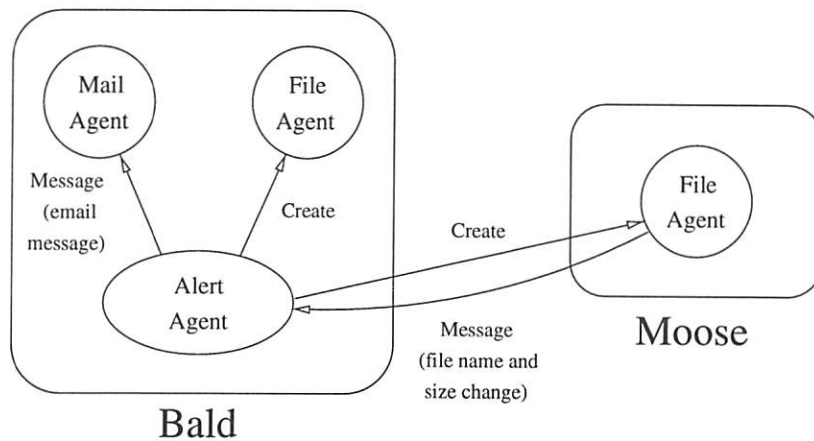
bald.cs.dartmouth.edu:
rgray  tty2      Sep  5 21:24 (:0.0)
rgray  tty6      Sep  7 07:14

cosmo.dartmouth.edu:
gvc     pts/0     Aug 23 10:11

...

```

Figure 3: The “who” agent submits a child agent that jumps from machine to machine and executes the Unix `who` command on each machine. The Tcl code is in the middle (the `agent` array holds the current location of the agent and is updated automatically as the agent migrates). The path of the agents through the network is shown at top. A fragment of the output appears at bottom.



```

set email_agent "bald rgray_email"      # machine and name of email agent
set machines "bald moose"
set directory "~rgray"

# get a name from the server
agent_begin

# submit the "file" agents that watch for changes in file size
for each m $machines {
    agent_submit $m -vars directory -proc file_watch {file_watch $directory}
}

# wait for one of the "file" agents to send a message saying that the
# status of a file has changed; then send an alert message to the user
# by asking the user's email agent to send a message to its owner

while {1} {

    agent_receive code string -blocking
    set alert [construct_alert $string]
    agent_send $email_agent {SEND OWNER $alert}

}

```

Figure 4: The "alert" agent monitors a set of files and sends an email message to the user when the status of a file changes significantly. A simplified version of the "alert" agent appears at bottom; procedure `file_watch`, which polls the files at regular intervals using the `file stat` command, and procedure `construct_mail`, which constructs a readable mail message, are not shown. The network location of the various agents is shown at top.

the user and then jumps to a permanently connected machine somewhere in the network. It then spawns one child agent for each collection. The child agents travel to the remote collections, perform the query using the available retrieval tools, and return to the permanently connected machine with the query results. The original agent then *discards all duplicates* and carries the results back to the mobile device. This approach allows the agent to carry on its retrieval work even when the mobile device is disconnected and minimizes the total number of bytes transferred across the low-bandwidth connection between the mobile device and the network (each document entry consists of the title, author and abstract so it takes only a few duplicates to add up to the agent's code size). In addition, there is no need to provide high-level search operations at each collection; since the child agents move to the collections, they can perform their search efficiently even if they must combine low-level primitives into the desired search operation.

Agent Tcl has also been used to retrieve three-dimensional drawings of mechanical parts from distributed CAD databases [CBC96], to track purchase orders [CGN96], and in several information-retrieval applications at external sites.

6 Future directions

The first area of future work is to finish the proposed architecture. We must add the hierarchical namespace, the nonvolatile store, and multiple languages and transport mechanisms. We are specifically interested in Java, Lisp, electronic mail and HTTP. Work on Java is in progress. In addition, we must finish the resource managers and add the security mechanisms that will protect an agent from a malicious machine and a group of machines from a malicious agent. Finally, we must extend our existing application agents so that they use the available security information.

The second area of future work is to add support agents. The resource managers that specify the security policies are one type of support agent. An effective mobile-agent system requires several more. We are in the process of identifying and constructing the necessary agents. Work on agents that provide directory services, navigation services, network-sensing tools, high-level communication services, and graphical construction tools is in progress.

The third area of future work is to experimentally compare the performance of mobile agents against traditional client/server solutions and to for-

mally characterize when an agent should remain stationary and when and how far it should migrate. Such a characterization must consider such things as network latency and bandwidth, relative machine speeds, code sizes, and data volumes.

7 Conclusion

Agent Tcl is a secure mobile-agent system that gains much of its flexibility and simplicity from use of the high-level scripting language Tcl. Although implementation work is not complete, Agent Tcl is in active use and has allowed the rapid development of efficient, distributed applications.

Availability

Agent Tcl version 1.2 will be available at <http://www.cs.dartmouth.edu/~agent> near the end of the summer; we are finishing the resource managers, writing the new documentation, improving the interface between Agent Tcl and PGP, and reworking the session-key implementation so that it does not require modifications to PGP. Agent Tcl version 1.1 is available now; version 1.1 uses Tcl 7.4, provides limited security and is somewhat slower. Agent Tcl runs on standard Unix platforms.

Acknowledgements

Many thanks to Professor David Kotz for reading the various incarnations of this paper and providing helpful criticism; to the anonymous reviewers for their constructive feedback; to Professor George Cybenko and Professor Daniela Rus for their support and encouragement; to Saurab Nog, Ting Cai, Yunxin Wu, Aditya Bhasin, Kurt Cohen and Scott Silver for their implementation work; and to the Air Force and Navy for their gracious financial support (ONR contract N00014-95-1-1204 and AFOSR contract F49620-93-1-0266).

References

- [BR] N. S. Borenstein and M. Rose. Safe Tcl. Available at <ftp://ftp.fv.com/pub/code/other/safe-tcl.tar.Z>.
- [CBC96] Kurt Cohen, Aditya Bhasin, and George Cybenko. Pattern recognition of 3D

- CAD objects: Towards an electronic yellow pages of mechanical parts. *International Journal of Intelligent Engineering Systems*, 1996. To appear.
- [CGH⁺95] David Chess, Benjamin Grosz, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. Technical Report RC 20010, IBM T. J. Watson Research Center, March 1995. Revised October 17, 1995.
- [CGN96] Ting Cai, Peter A. Gloor, and Saurab Nog. DartFlow: A workflow management system on the web using transportable agents. Technical Report PCS-TR96-283, Department of Computer Science, Dartmouth College, May 1996.
- [Coe94] Michael D. Coen. SodaBot: A software agent environment and construction system. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.
- [Gra95] Robert S. Gray. Agent Tcl: A transportable agent system. In James Mayfield and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.
- [Har95] Kenneth E. Harker. TIAS: A Transportable Intelligent Agent System. Technical Report PCS-TR95-258, Department of Computer Science, Dartmouth College, 1995.
- [JvRS95] Dag Johansen, Robbert van Renesse, and Fred B. Scheidner. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.
- [KK94] Keith Kotay and David Kotz. Transportable agents. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.
- [KPS95] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice-Hall, New Jersey, 1995.
- [LO95] Jacob Y. Levy and John K. Ousterhout. Safe Tcl toolkit for electronic meeting places. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 133-135, July 1995.
- [NCK96] Saurab Nog, Sumit Chawla, and David Kotz. An RPC mechanism for transportable agents. Technical Report PCS-TR96-280, Department of Computer Science, Dartmouth College, 1996.
- [Ous95] John K. Ousterhout. Scripts and agents: The new software high ground. Invited Talk at 1995 Winter USENIX Conference, January 1995.
- [Pei96] Holger Peine. The ARA project. WWW page <http://www.uni-kl.edu/AG-Nehmer/Ara>, Distributed Systems Group, Department of Computer Science, University of Kaiserslautern, 1996.
- [Sah94] Adam Sah. TC: An efficient implementation of the Tcl language. Master's thesis, University of California at Berkeley, May 1994. Available as technical report UCB-CSD-94-812.
- [Sun94] The Java language: A white paper. Sun Microsystems White Paper, Sun Microsystems, 1994.
- [TV96] Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the 41th International Conference of the IEEE Computer Society (Comp-Con '96)*, February 1996.
- [Way95] Peter Wayner. *Agents Unleashed: A public domain look at agent technology*. AP Professional, Chestnut Hill, Massachusetts, 1995.
- [Wel95] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, New Jersey, 1995.

- [Whi94] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc., 1994.
- [Wu95] Yunxin Wu. Advanced algorithms of information organization and retrieval. Master's thesis, Thayer School of Engineering, Dartmouth College, 1995.

TclJava: Toward Portable Extensions

Scott Stanton

Sun Microsystems Laboratories
sstanton@eng.sun.com

Ken Corey

Sun Microsystems Laboratories
kcorey@eng.sun.com

96-0149

Abstract

This paper discusses an extension to Tcl that allows the programmer to define Tcl commands in Java code, evaluate Tcl scripts and use the Tk graphical user interface from Java. This might be used to recode performance critical parts of Tcl scripts, provide user configuration of an application for Java, or offer other capabilities. The source code is available, see the 'Sources' section at the end.

Introduction

Tcl [Ousterhout94] was originally created to add scripting capabilities to C programs. The design of Tcl makes it easy to add scripting to C applications by defining new Tcl commands in C.

However, as more applications are sent over the Internet, the need for completely portable applications is increasing. Since the Tcl language has already been ported to Microsoft Windows, Macintosh, and many Unix variants, the Tcl parts of hybrid C/Tcl applications are already portable. Unfortunately, C does not directly support cross platform execution without recompilation.

Fortunately, Java, a portable, compiled language similar to C++ [Hamilton96], can easily fill the same role C code does for Tcl, but in a portable way.

What can Java provide?

Java complements Tcl in ways similar to C. Java is a compiled language, so it is possible to write compute intensive functions in Java, while keeping the user interface in Tcl/Tk.

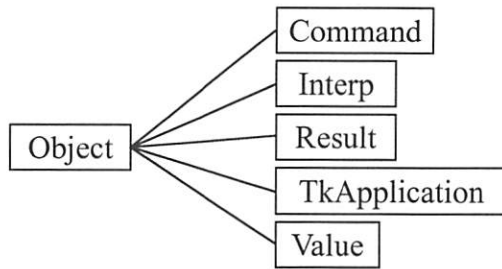
Further, because Java is compiled, it can provide code obfuscation for organizations wanting the benefits of Tcl but who are reluctant to distribute source code. Java is portable, so it is possible to pass the compiled object files ('.class' files) between different machines, and be sure that they work on all platforms. Java provides support for data structures within an object-oriented framework.

How does Tcl fit Java?

Tcl complements Java in several ways as well. It provides a friendly yet powerful user interface that is portable between different platforms. It eases maintenance of legacy code by allowing changes to an interface very quickly because of Tcl's rapid development cycle. It also allows for ease of configuration in Java programs. These two languages work quite well in different spheres. When combined, they work even better.

Approaches

We examined three approaches for the TclJava interface. The first approach, developed by Patrick Naughton and Arthur van Hoff [Naughton94], forced the programmer to sub-



Roughly equivalent to the 'Interp' structure in C.

Equivalent to the result member of the 'Interp' struc

Figure 1. The TclJava Class Hierarchy, and roughly equivalent structures in the C interface.

class the interpreter class to add new commands. This inheritance-based approach, though familiar and obvious from a C++ programmer's viewpoint, had several problems:

- Static structure, not allowing commands to be dynamically added
- Requires a recompile to add a new command to an interpreter
- Cannot delete a command from the Tcl interpreter

The second approach we considered was a completely string based implementation, where the name of the object and the method to invoke were constructed dynamically by the TCL script. This approach had the advantage of being very dynamic and flexible. However, this approach had problems as well:

- Does not provide type-safety
- Allows malformed strings to call random or nonexistant Java methods
- Does not fit well with the Java programming model

Our third design used a composition based model for defining Tcl commands in Java. This approach allows commands to be added and removed from an interpreter while avoiding unsafe dynamic dispatches. Type checking is maintained, while dynamicism is allowed. The composition design is described in more detail in the following sections.

An Interpreter in Java

The TclJava interface is comprised of several Java classes that encapsulate a subset of the Tcl C API. These classes are shown in Figure 1, and explained in detail below.

In C code that accesses the Tcl API's, the programmer must keep track of a 'Tcl_Interp' structure. This structure encapsulates the state of the given interpreter. When using the TclJava interface, a Java class called 'Interp' contains a reference to the C 'Tcl_Interp' structure, in addition to several methods used to interact with an instance of the 'Interp' class. This instance is passed to many of the methods of the various objects. There can be many interpreters in one Java program, so it is necessary to indicate which one to many of the methods on the various objects shown above.

Evaluating a Tcl script from Java is similar to doing it in C. To execute a Tcl command in C, the programmer calls the Tcl_Eval function which returns the completion code. In Java, the programmer calls the 'Interp.eval' method, which returns a completion code. The completion code is the numeric result indicating the success or failure of the command.

In addition, the Java 'Result' object contains both the numeric result and the string result of the last command. In C, the command procedure puts the string component of the result in the 'result' field of the 'Tcl_Interp' structure, and returns the numeric status code. In Java,

the 'Result' object contains both of these items: a numeric component to indicate if the command executed properly, and a 'Value' component to contain the string result of commands.

A new Tcl command is created from Java by defining a subclass of 'Command' that implements the 'invoke' method. The 'Command' object is equivalent to the command procedure and the client data argument used in the C interface.

The body of a callback is shown below. The programmer subclasses the abstract class 'Command', placing the body of the command in the 'Command.invoke' method as below. Note how the 'Command.invoke' procedure sets the string result using the method 'interp.setResult', and then returns the status:

```
class Callback extends Command
{
    public int invoke(Interp
        interp, Value
        args[]) {
        /*
         * body of the command
         * here
         */
        interp.setResult(new
            Value("The result
                string to return.));
        return Interp.TCL_OK;
    }
    public void deleted(Interp
        interp) {
        /*
         * code to execute when
         * the
         * command is deleted
         * from the interpreter.
         */
    }
    protected void finalize() {
        /*
```

```
         * code to execute when
         * this
         * java command object
         * is finalized (garbage
         * collected)
         */
    }
}
```

Figure 2. Defining a callback object for a new command from within Java.

There are three main steps to writing a TclJava program. First, the Java application creates an Interp object, and initializes any other Tcl extensions that are to be used. Tk is the only such extension provided in this interface at this time. The 'TkApplication' class provides a method to initialize the Tk extension, and a simple event handling loop to handle user interaction. No other hooks into the Tk API's are provided, so the Java code must use the 'Interp.eval' method to control or query the Tk environment.

```
class myClass {
    /*
     * Other methods may be
     * defined
     * here.
     */
    public static void
        main(String args[]) {
        Interp interp = new
            Interp();
        TkApplication.TkI-
            nit(interp);
        .
        .
        .
    }
}
```

Figure 3. Beginning a Java program that uses Tcl/Tk.

After creating the interpreter, the application uses the 'Interp.createCommand' method to create a new Tcl command that is implemented by the specified Command object.

```
interp.createCommand("new-
```

```

command\ name", new
Callback() );
.
.
.

```

Figure 4. Creating a Tcl command from Java.

Finally, the programmer enters the event loop for the TkApplication. Note that in Java programs that use AWT, this event loop is implicit. When the TclJava application enters the 'mainLoop' method, it won't return until all the Tk windows are destroyed.

```

TkApplication.mainLoop();
}
}

```

Figure 5. Finishing the TclJava program.

Limitations of this interface

Although the TclJava interface exposes most of the functionality needed to implement interesting applications using Tcl/Tk and Java, it still has a number of limitations:

- Passing arguments between Tcl and Java requires a fresh allocation and de-allocation of a Java array, and a copying of the C strings to Java Strings each time the callback is invoked. This results in the garbage collection system being called quite often. Because of these efficiency concerns, Tcl extensions written in Java should minimize the amount of data passed across the TclJava interface. In the example program, a file name is passed to the Java-defined command, and a single string (consisting of all the data in a list format) is returned. The traffic over the TclJava interface is kept to a minimum. One could imagine another design, where each element of the zip file were returned one at time. This approach could perhaps provide for more detailed control, but would also lower the efficiency of the interface.
- Most of the C API's provided by Tcl/Tk

aren't exposed to Java. The only way for TCL scripts to interact with the Java binaries is through command implementation callbacks. The only way for Java to access the state of the interpreter is through the 'Interp.eval' method. To make TclJava more complete, many of the existing C API's should be added.

- No link exists between Java variables and Tcl/Tk variables. Tcl provides interesting tracing capabilities on variables that are not available from Java.
- Java's graphical user interface, AWT, and Tk are mutually exclusive. One cannot currently write applications that utilize both interfaces.
- Tcl/Tk is not thread safe, so it is very important that all calls that do anything with the Tcl interpreter come from a single thread. If calls are made from different threads Tcl/Tk will crash, usually quickly.

Conclusion

This interface is a convenient way to extend the capabilities of both languages in way that is portable between the Unix, Macintosh and Windows worlds. Using composition instead of inheritance to add new commands to a Tcl interpreter allows for a dynamic environment in Java while maintaining the type safety of the Java objects. Given the minimalist nature of the TclJava interface, this work should be viewed as a starting place for further Tcl/Java integration.

Acknowledgments

Thanks to John Ousterhout, Chris Perdue, Steve Uhler and Brian Lewis for making valuable suggestions on short notice.

References

[Ousterhout94] John K. Ousterhout, (ouster@eng.sun.com), Writing Tcl Applications in C, Tcl and the Tk Toolkit, July 1994.

[Naughton94] Patrick Naughton, (naughton@starwave.com), Java/Tcl Interface. Sun Microsystems, Inc, World Wide Web page <http://java.sun.com/applets/alpha/notapplets/tcl/>, November 1994.

[Hamilton96] Mary Hamilton, Java Documentation. Sun Microsystems, Inc., World Wide Web page <http://java.sun.com/doc.html>, December 1995.

Sources

All source code mentioned in this paper is available online as <ftp://ftp.smli.com/pub/tcl/tcljava#.#.tar.gz>, where ‘#.#’ is the version number.

A Tk Netscape Plugin

Jacob Levy

Sun Microsystems Labs

2550 Garcia Ave., Mountain View, CA 94043

Abstract: I have built a plugin module for Tk, for Netscape Navigator. This module delivers Tk applications as elements of Web pages and it makes it possible to create web based applications that have a richer GUI than HTML. Tcl also makes building such applications easier due to its high level scripting nature, and Safe Tcl helps construct web applications that can safely perform interesting tasks such as communicating with local or remote resources. The demonstration will show how the plugin module works and some of its capabilities and limitations.

1.0 Introduction.

I believe that Tcl and Tk [1] are useful for scripting the Internet. Tcl provides an expressive set of features that are portably implemented on a wide variety of platforms. Tcl has recently been enhanced with network communication capabilities, significantly easing the task of creating network aware applications. Tk is a mature and portable GUI kit built on top of Tcl that is available for all of today's popular desktop systems.

To make Tcl a ubiquitous scripting language for the Internet, a delivery vehicle for Tcl applications over the network is needed. Netscape Navigator [2], with its ability to host plugin modules [3], offers such a delivery mechanism. A plugin module is an extension, supplied by a third party, to allow a browser to display information for which the browser does not have built in support. A plugin module displays its data in a window given to it by

the browser. The plugin module can interact with the user through this window.

Plugin modules are not complete programs; they are activated inside the browser's address space and windowing hierarchy. While there is no standard in this area yet, generally the browser limits interaction between a plugin module and other displayed elements of a page. Plugin modules are written in C/C++ and have full access to the hosting system.

Embedding Tk programs in Web pages provides unique advantages:

- Tk allows the construction of much richer and reactive GUIs than possible with HTML [4]. For example, in HTML there is no way to display structured graphics, as is possible with Tk's canvas widget.
- Tcl, as a high level scripting language, facilitates the development of interactive web based applications. It is much easier to write a Tk program to perform a web based function than it is to write a complete plugin that delivers the same functionality.
- Safe Tcl [5] provides a solid base for flexible security policies that allow applications to perform functions that would otherwise be impossible in untrusted scripts. For example, Safe Tcl can provide security policies that allow varying degrees of access to local and remote resources. The Tcl plugin can make use of Navigator's authentication facilities, when Navigator will be extended with the required APIs.

- Tcl, due to its late binding, allows incremental delivery of applications over the Web. An application can be partitioned into small chunks of functionality that are loaded on demand, as the user requests access to each part. Additionally, Netscape Navigator's cache policy can be configured to ensure that the user always sees the latest version; this eliminates the need to maintain several versions of an application.

Two goals were paramount in designing the Tk plugin module:

- Tk applications should operate identically whether they are embedded inside a Web page or not, and whether they are delivered over a network or are locally available. This immediately makes all existing Tk applications deliverable via the plugin. The “.” window should behave the same irrespective of whether the application runs inside the plugin module. Also, the full power of Tk, including event driven programming, should be available in the plugin module.
- Tk applications should execute in a safe environment, so that they cannot harm the user visiting a page. On the other hand, this environment should not be too restrictive, otherwise many interesting applications are ruled out.

The paper describes the implementation issues I faced when constructing the plugin module. It then describes some possible uses of the Tk plugin module, and discusses what the plugin can and cannot do. I then describe the current status of the development of the plugin, and briefly hint at possible future projects.

2.0 Implementation Issues.

To implement the Tk plugin module, I encountered three main issues:

- Embedding Tk windows inside another application's window hierarchy.
- Preserving Tk's event model as well as Netscape's event handling.
- Providing a safe yet functionally rich execution environment for untrusted scripts.

Netscape Navigator provides a native window in which an instance of a plugin can display itself. The window belongs to Netscape and can be modified at will by Navigator. Navigator provides notifications to the plugin to inform it of important events that affect the window such as resizing and destruction of the window. However, Tk expects that all windows that it uses to display content are created by it and are completely owned by it. Tk also believes that “.”, the application's main window, is really a toplevel window and not the child of some other window; Tk wants to set the size and position of toplevel windows and interact directly with a window manager. To address this, Tk has been enhanced so that it is able to create a main window that is a child of another window. This also provides Tk with the illusion that it is interacting with a window manager to manage the position and size of its main window.

A related issue is event handling. Tcl has its own event loop that provides file and timer based events, in addition to UI events. Because I do not have access to the event loop in Navigator, I decided to provide a separate event loop for the plugin module, using the most appropriate mechanism for each platform. This also avoids extensive modifications to Tk. In Win32 [6] and Solaris 2.x, I use a timer that causes the Tcl event loop to be entered periodically. This allows the plugin module to react to events in a timely fashion without interacting with Navigator's event loop. I considered running Tk inside its own process and communicating with Navigator using sockets; however it is difficult to manage a window

belonging to one process inside another, and the plugin would not have the same level of access to Navigator's API as is possible when it executes within the same address space.

Incoming scripts are not trusted and should be executed in an environment that prevents them from damaging the host system by e.g deleting files or stealing private information. Safe Tcl provides an environment in which an untrusted script cannot do such damage. Each script is executed inside its own interpreter which is made safe by removing "dangerous" commands. Safe Tcl also provides a mechanism for implementing security policies that allow untrusted scripts to safely perform many interesting tasks: interpreters are arranged in a master slave hierarchy; a slave interpreter can be extended by its master with safe access to unsafe functionality in the master. For example, a policy can enforce that a script has read only access to local files but cannot communicate with off-site processes.

Tk prevents a script from damaging the host site or stealing its private information; however, currently it allows scripts to mount denial of service attacks, e.g. by globally grabbing the mouse and never releasing the grab. A subset of Tk that prevents such attacks should be defined and be made available in safe interpreters. This is an area for further research.

3.0 Uses for a Tk Plugin Module.

Three main uses are envisioned for the Tk plugin module:

- Provide richer GUI elements in a page than is possible with HTML alone. Tk can be used, e.g., to decorate pages with interactive structured graphics using canvas widgets that provide bindings for events to interact with the user in interesting ways.
- Implement interactive forms that provide client side validation and assistance; for example, a tax form application can perform

internal consistency checks and suggest allowable deductions that the user did not take. An application can be structured as a series of linked forms; for example, if the user selects a home mortgage interest deduction in the main form, she can be presented with an itemized deduction schedule to fill out in another form. The script can also automatically fill out parts of the form for the user if it is given access to local data such as the user's email address.

- Safely host applications that require communication with a variety of backends such as databases, or access to local resources. Tcl provides its own communication mechanisms such as sockets that can be safely used by untrusted applications, and it allows a range of interesting security policies to govern access to local and remote resources. For example, a script can connect to a remote multi-user forum and provide a way for the user to interact through its window with other users in a shared application. As another example, a user can safely give a script access to her private financial data using a security policy that prevents the script from communicating remotely.

4.0 Limitations of the Plugin.

There are some limitations to what a plugin module can do; these mostly stem from the very narrow interface provided by the hosting browser. For example, a plugin module currently does not have access to other elements of the displayed page such as embedded forms. Some limited access to JavaScript [7] is possible, but it is currently not possible to control Java [8] applets or other plugins from inside a plugin. While access to forms written in HTML is desirable, Tk is rich enough to allow a form to be constructed entirely with Tk UI elements, but this does require programming.

5.0 Status and Future Plans.

I intend to release the plugin for Windows NT, Windows 95 and Solaris 2.x at the workshop, for experimentation by the Tcl community. A MacOS version of the plugin will be distributed later. The plugin is in active development at the time of writing and its capabilities are improving rapidly. It is currently able to display Tk applets inside windows embedded in Navigator's window hierarchy, and it is able to load scripts from remote or local URLs.

ActiveX [9, 10] is being promoted by Microsoft as an alternative to plugins. I have created a version of the plugin packaged as an OLE control (OCX) [11], and I will also release this package at the workshop. My intent is to continue developing the OCX to make it usable as a component in the upcoming Windows 95 desktop which will be an OCX container.

Plugin modules are currently not usable in a variety of browsers, because there is no standard for the interface between the browser and the plugin module. I expect that increasing interest in plugin modules will spur standardization efforts in this area, perhaps under the auspices of the W3 consortium. Therefore I will target my efforts to provide a Tk plugin module at those plugin enabled browsers that capture a significant market share. I expect that more browsers will shortly support plugin modules. Microsoft Internet Explorer [12], Spyglass Mosaic [13] and Oracle's PowerBrowser [14] also support plugins; Oracle PowerBrowser uses Netscape's plugin API.

I am planning to add the following functionality to the plugin, to enable it to host more powerful applets:

- The ability to send data to Navigator for display. This will allow the plugin to behave as a filter or HTML generator.

- The ability to initiate requests for new remote or local URLs to be fetched and presented to the applets as Tcl channels. This will enable the plugin to host applets that are composed of several parts, and not require that they all be downloaded before any display can occur. The main problem in realizing this is likely to be how to allow this while still preserving the privacy of information in the user's environment.
- The plugin should allow each applet to manage more than a single toplevel window within the page being displayed by Navigator. This will make possible the construction of powerful form-like interfaces with interactive behavior, e.g. for data verification and consistency checking. Of course, because the full power of Tk is available in each of these toplevel windows, the interface is not limited to only form elements.

6.0 Acknowledgments.

Early versions of the plugin module were developed by Jeff Hobbs. Colin Stevens provided Win32 expertise and helped develop the plugin module on Windows NT and Windows 95. John Ousterhout made the enhancements to Tk that allow it to be embedded in other window hierarchies.

7.0 References.

1. "Tcl and the Tk Toolkit", John K. Ousterhout, Addison-Wesley, (1994).
2. Netscape Navigator 3.0, URL: <http://home.netscape.com/>.
3. Netscape Navigator SDK, URL: <http://home.netscape.com/>
4. W3 Consortium Web Site, URL: <http://www.w3.org/pub/WWW/>.

5. "A Safe Tcl Toolkit For Electronic Meeting Places", Jacob Levy, in Proc. 1st USENIX Workshop on Electronic Commerce, pp. 133-137, NY, NY (1995).
6. "Win32 Programmer's Reference, Vol. 1-5", Microsoft Press, (1994).
7. JavaScript URL: <http://www.netscape.com/>
8. Java URL: <http://www.javasoft.com>.
9. ActiveX URL: <http://www.microsoft.com/intdev/sdk/>.
10. "Inside OLE" Kraig Brockschmidt, Microsoft Press (1995).
11. "OLE Controls Inside Out", Adam Denning, Microsoft Press (1995).
12. Microsoft Internet Explorer, URL: <http://www.microsoft.com/windows/ie/msie.htm>.
13. SpyGlass Mosaic, URL: <http://www.spy-glass.com/>.
14. Oracle PowerBrowser, URL: <http://www.oracle.com/products/websystem/powerbrowser/html/index.html>.

TclDG - A Tcl Extension for Dynamic Graphs

John Ellson
john.ellson@lucent.com
Stephen North
stephen.north@att.com

Abstract

TclDG is a toolkit extension for the manipulation, rendering, and interaction with abstract graphs and dynamic graph views. This paper introduces TclDG by examples that progressively build up to an implementation of a multiple-view graph editor.

1 Introduction

Drawings of abstract graphs and networks are an important component of user interfaces that focus on relationships or connections between objects. Effective techniques for viewing static graphs are fairly well understood [RDM⁺87, Him89, PT90, GKNV93], but many applications could benefit from dynamic graph layouts. Dynamic layouts are needed when the underlying data modeled by graphs can change. Some plausible examples include communication networks whose connectivity changes, and dynamic data structures displayed as graphs in an interactive debugger. Dynamic layouts are also especially relevant if the user can adjust the set of visible objects for browsing graphs or networks too large to be drawn in their entirety, such as linked WWW documents [Dom95].

Besides dynamic displays, another key property of advanced graph browsers is that they be easily customized and extended to handle application-specific features. This suggests a toolkit approach.

TclDG is a practical user interface toolkit that incorporates incremental graph layouts. It was written specifically for prototyping user interfaces to distributed network management systems. TclDG is a direct descendent of TclDot [EN95]. Where TclDot was built on the same graph libraries as the Unix tool “dot,” TclDG is built on the next-generation of those graph libraries. It retains much stylistic compatibility with TclDot but its incremental layout mechanisms and its technique for coupling with Tk’s canvas are quite different.

Figure 1 shows three interactive views of a

sample graph. Users can insert or delete nodes or edges in any view and the resultant graph is updated in all views. Though different layout algorithms are employed, the views are kept in synchrony to depict the same base objects. The 100-line TclDG script that implements this three-view graph editor is discussed in this paper.

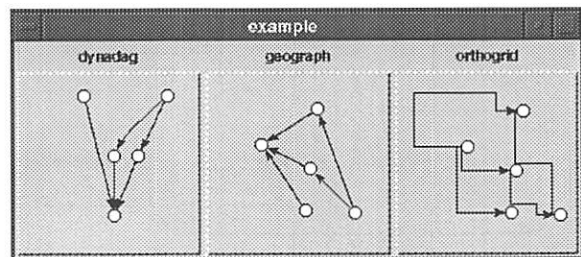


Figure 1

2 Architectural Framework of TclDG

The conceptual framework of TclDG combines graphs, views, and canvasses:

A graph is a set of nodes and edges data structures maintained by Libgraph, but without necessarily any intrinsic positional properties. The sets can be nested into subgraphs, and all items (nodes, edges, graphs, and subgraphs) can have arbitrary application-defined attributes.

A view is also a graph, but one which has been annotated with positional information generated by a layout engine. A view in TclDG does not provide support for application defined attributes since these can be stored in a graph and might be usefully shared between multiple views.

Tk’s canvas provides a rendering of a view. Nodes and edges may be further decorated on the canvas. TclDG node and edge handles can be used as tag values with the associated canvas items allowing user events to be related back to the abstract objects represented by the graph. There is no hard de-

pendency on Tk's canvas, so other rendering devices can be used instead. Two useful ones are: GIF-Draw (gd1.2 with tclgd [Bou95, Tho95]) and Pad++ [Con95].

TclDG allows multiple views per graph and multiple canvasses per view. Also, the graph data structure facilities of TclDG can be useful in applications without any views or canvasses for maintaining graph data structures in non-graphical applications.

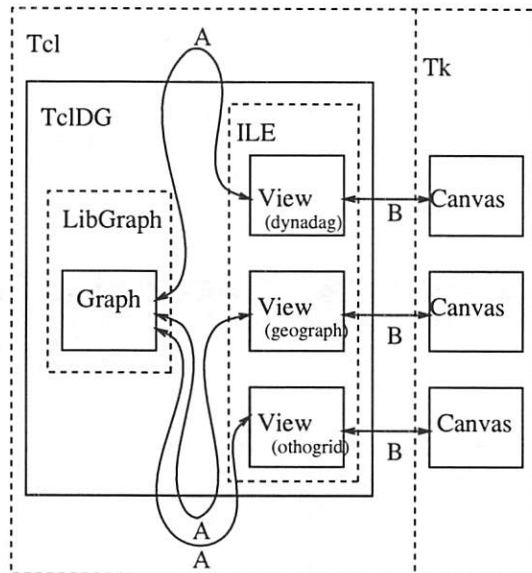


Figure 2

Figure 2 diagrams the architecture of TclDG as instantiated in the example presented in this paper. TclDG provides a Tcl script-level interface to instances of graphs, supported by Libgraph, and to instances of views, supported by the various Incremental Layout Engines (ILEs).

One, possibly novel, aspect of this architecture is the decoupling of the abstract graphs from views. At the level of the C libraries, ILE does not depend on the Libgraph interface, instead ILE is a "black box" interface employing 32 bit identifiers embedded in succinct descriptors for layout elements. The descriptors contain geometric coordinates and some related fields. The namespace of identifiers is controlled by clients; IDs usually refer to client-side descriptors. Engines inform clients of changes to views by invoking callback functions that pass descriptors. Our original design did explicitly layer ILE on Libgraph, but we found it convenient to allow ILE clients to define their own graph data structures.

TclDG provides a "handle" for each node (or edge) in the main graph that can be used in the views, and as a tag on canvas items.

The Tcl script writer has control over all interactions between graphs and views (A), and views and canvasses (B), because interconnection is programmed by Tcl methods and bindings, not hard-wired. In the example script presented in this paper the Tcl processing between graphs, views, and canvasses is minimal, but in a more advanced TclDG application these interfaces can be quite valuable. For example, the underlying graph can be larger (contain more nodes and edges) than any individual view; the scripts at points (A) can filter the graph according to some application defined predicate for inclusion in the view. This can be used to produce say: inheritance diagrams, include file dependencies, and ER diagrams, all from a common graph representing a set of source code.

In the example graph editor application, graphs, views, and edges are "connected" by bindings such that user events propagate from the user, through one view, to the graph and from there distributed back out again to the user via all the views. In this way, all the views are updated to be consistent with the underlying abstract graph, although they may use different layout engines and offer other stylistic differences in rendering.

3 Graphs

In this section we describe the methods and bindings of graphs. The next section will provide a similar introduction to views; then we will combine these elements to present an example graph editor script.

To reiterate, a graph is a set of nodes interconnected by edges. Nodes can be added and deleted from the graph, and edges can be added or deleted between pairs of nodes. String attributes of nodes and edges can be set and modified. Primitives are supplied for editing node and edge sets and defining and changing attributes. A detailed description of each command can be found in the TclDG man page. Here we introduce a selected subset of the methods by means of a snapshot of an interactive session with TclDG.

3.1 Graph, node and edge construction

The interpreter is started from the shell with the `tclldg` command that produces an unmodified `tcsh` prompt. `TclDG` may also be dynamically loaded into an unmodified `tcsh` executable.

```
$ tclldg
%
```

A new empty graph is created with the `dgnew` command that returns a handle to the new graph. Graphs can be directed or undirected, strict or not strict. A strict graph is one with no self-edges or parallel edges, a directed graph is one in which an a-b edge is considered different than a b-a edge.

```
% set g [dgnew digraph]
graph0
```

Nodes can be added to the graph with the `addnode` method that returns the handle of the new node.

```
% set n [$g addnode]
node0
```

All handles in `TclDG` are also registered as new `Tcl` commands, so handles are typically assigned to variables that are then used with method parameters to perform operations on the object.

```
% set m [$g addnode]
node1
% set p [$g addnode]
node2
```

Similarly edges can be added to the graph with `addedge`, but since node handles are also registered as commands edges can alternatively be added by using the slightly shorter `addedge` method of nodes.

```
% set e [$g addedge $n $m]
edge0
% set f [$m addedge $p]
edge1
```

3.2 Graph, Node and Edge Attributes

Nodes and edges can have user defined attributes whose values are strings. Once an attribute name has been declared for one node, then all nodes have an attribute of that name.

```
% $n set a 3
3
```

There is a similar namespace for edge attributes.

```
% $n set a
3
```

In real applications attributes usually play a crucial role in supporting application-specific graph semantics.

```
% $m set a
%
%
```

3.3 Introspection and Navigation of a Graph

Methods are provided for several kinds of graph searches and traversals. Here is a small sampling of the methods.

Methods that return handles are convenient for use in compound statements like

```
[$e headof] listoutedges
```

Methods that return lists of handles are convenient for use in `foreach` loops.

```
% $g listnodes
node0 node1 node2
% $g listedges
edge0 edge1
% $m listinedges
edge0
% $m countoutedges
1
% $e tailof
node0
```

3.4 Deletion

Nodes, edges, and graphs all have a `delete` method. `% $f delete`
If a node is deleted then all subtending edges are also `% $n delete`
deleted. If a graph is deleted then all the nodes and `% $g listnodes`
edges of the graph are deleted along with it. `node1 node2`
`% $g listedges`
`%`

3.5 Graph Bindings

Bindings in TcIDG are analogous to the bindings in Tk, except that these support a different set of events completely independent of X events. Graph bindings are provided to allow views to be notified of events that occur in the graph, such as a node insertion or an attribute modification. All the bindings in TcIDG share a common set of "%" substitution parameters. These are:

`%g` The graph handle of the graph generating the event.
`%v` The view handle of the view generating the event.
`%n` The node handle of the node generating the event.
`%e` The edge handle of the edge generating the event.
`%A` The attribute name (or shape type)
`%a` The attribute value (or boundary point list).

Not all of these are appropriate in all TcIDG events - an empty string is supplied for parameters that cannot be substituted.

Here we set up the three node bindings and illustrate the resulting callbacks by generating each of the event types: insertion, modification, and deletion.

```
% $g bind insert_node {puts "insert: %g %n"}
% $g bind modify_node {puts "modify: %g %n %A %a"}
% $g bind delete_node {puts "delete: %g %n"}
% set n [$g addnode]
insert: graph0 node0
node3
% $n set a 9
modify: graph0 node3 a 9
9
% $n delete
delete: graph0 node3
```

Multiple bindings may be attached to the same event. This can connect a graph to multiple views, or a view to multiple canvasses. Additional bindings are indicated by a leading "+" in the binding string.

```
% $g bind insert_node {+puts "insert2: %g %n"}
% $g addnode
insert: graph0 node3
insert2: graph0 node3
node3
```

The full set of bindable events can be found by the introspection mode of the `bind` method, i.e. without any parameters. The current bindings for any event can also be determined.

```
% $g bind
insert_graph modify_graph delete_graph
insert_node modify_node delete_node
insert_edge modify_edge delete_edge
% $g bind insert_node
puts "insert: %g %n"
puts "insert2: %g %n"
```


Debugging trick: this one-liner binds a debug state-
ment to all the possible events of a graph.

```
% foreach b [$g bind] {$g bind $b
"+puts \"$b %g %e %n %A %a\""}}
```

4 Views

A "view" is a specialized data structure supported by the ILE library. Objects in views (nodes, edges, and higher-order structures) are represented by descriptors that contain an object ID and associated geometric coordinates. The principal operations in ILE are:

- open or close a view
- insert a new object
- modify an object's position or shape
- delete an object
- hold or release callbacks

(The operations to hold and release callbacks are not exposed to the Tcl programmer but used by TclDG to provide atomic operations.) A few other functions assist with mapping between IDs and descriptors and walking the contents of a view.

The set of layout engines is extendible. The selection includes:

- DynaDag - Hierarchical directed graphs, splined edges [Nor96]
- OrthoGrid - Incremental manhattan layout [MHT93]
- GeoGraph - User-defined node placement, straight edges. (For graphs in which nodes are manually placed, or in which nodes have intrinsic location properties that dictate placement)

Figure 3 shows two successive frames in the execution of DynaDag. The frame on the right shows the result of incrementally adjusting the layout on the left, after adding an edge from the rightmost to the middle node on the second level.

4.1 Node and Edge Insertion into Views

When a node is inserted into a view, information about its shape (bounding polygon), and an optional initial coordinate (mouse click hint) are provided to the view's engine. The shape descriptor permits computing the correct separation between center points of adjacent nodes and in clipping edge splines or polylines to the node's boundary. (In a simple graph editor [Ous94], one usually draws the edge to the center of the node and obscures the end by drawing the node shape on top. However this technique does not lend itself to drawing arrowheads on directed edges.)

In this fragment we insert node `$n` as an oval, and node `$m` as a triangle. Note that the supported shape names and coordinate syntax is designed to be compatible with Tk's canvas items.

```
% set v [dgview dynadag]
view0
% $v insertnode $n oval {-10 -10 10 10}
% $v insertnode $m polygon {-10 10 0 -10 10 10}
% $v insertedge $e
```

4.2 View Introspection

View introspection methods are helpful in situations in which a predicate for node or edge inclusion in a view depends on the nodes and edges already inserted.

```
% $v listnodes
node0 node1
% $v listedges
edge0
% $v isnodeinview $n
1
% $v isedgeinview $f
0
```

4.3 View Deletion

As in graphs, deletion methods release resources associated with objects in views.

```
% $v deleteedge $e
% $v deletenode $m
% $v delete
```

4.4 View Bindings

View bindings operate similarly to graph bindings, but view operations such as adding an edge may yield multiple callbacks as the layout engine adjusts other nodes and edges around the new one.

In insert and modify bindings %A and %a substitutions provide shape and coordinates in the format expected by the canvas create operations.

Callbacks for deletion events are issued just prior to deallocation.

```
% $v bind
insert_node insert_edge modify_node
modify_edge delete_node delete_edge
% foreach b [$v bind] {
    $v bind $b "+puts \"$b %v %e %n %A %a\""
}
% $v insertnode $m oval {-10 -10 10 10}
insert_node view0 node0 oval -10 0 10 20
% $v deletenode $n
delete_node view0 node0
```

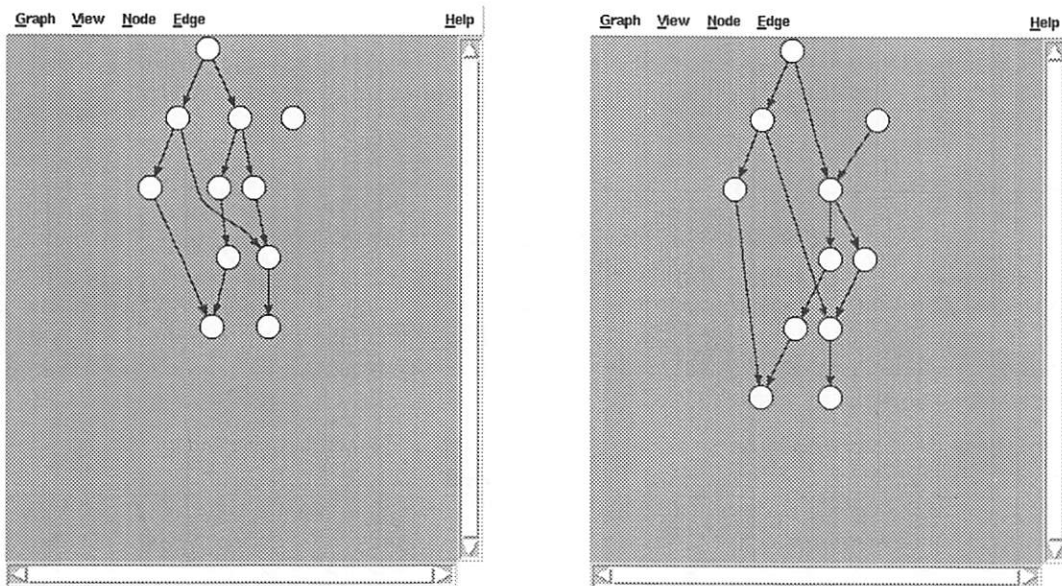


Figure 3

5 Putting it all together

This is the script for the three view graph editor shown in Figure 1. The bulk of the code handles user input events. The most interesting section of the code is the `newview` proc, in which connections are established between views and graphs, and between graphs and canvasses.

```
#!/usr/add-on/tcl7.5/bin/tclsh
load /usr/add-on/tcl7.5/lib/libtcltdg.so
load /usr/add-on/tcl7.5/lib/libtk.so

# example multi-view graph editor
#
# mouse button 1 inserts node, or edge if pressed over an existing node.
# mouse button 3 deletes node or edge under mouse
#
# insert or delete in any view, result is displayed in all views.

# set to 1 to trace graph and view events
set debug 0

# define shape for nodes
set shape oval
set boundary [list -5 -5 5 5]

# set global x,y according to canvas-relative mouse position
proc adjust_xy {c wx wy} {
    global x y
    set x [$c canvasx $wx]; set y [$c canvasy $wy]
}

# find a graph object under the mouse of one of the types
# given in $args, the handle of the object is left in $obj
proc find_obj {c args} {
    global x y
    upvar 1 obj obj
    foreach item [$c find overlapping $x $y $x $y] {
        foreach obj [$c gettags $item] {
            foreach type $args {
                if {[string first $type $obj] == 0} {return 1}
            }
        }
    }
    set obj {}
    return 0
}

# start a node or edge insertion, if started over a node then insert edge
proc b1_down {c} {
    global oldx oldy x y shape boundary id tail
    if {[find_obj $c node]} {
        set id [$c create line $x $y $x $y -fill red]
    } {
        set id [$c create $shape $boundary -fill red]
        $c move $id $x $y
    }
    set oldx $x; set oldy $y
    set tail $obj
}

# move node before completing insertion, or extend edge
proc b1_move {c} {
    global oldx oldy x y id tail
    if {$tail != {}} {
        $c coords $id $oldx $oldy $x $y
    } {
        $c move $id [expr $x-$oldx] [expr $y-$oldy]
        set oldx $x; set oldy $y
    }
}
```

```

    }
}

# complete node or edge insertion
# x,y are left in global vars for use by the event bindings
proc b1_up {c g} {
    global id tail
    $c delete $id
    if {$tail != {}} {
        if {[find_obj $c node]} {$tail addedge $obj}
    } {
        $g addnode
    }
}

# delete a node or edge
proc b3_down {c} {
    if {[find_obj $c node edge]} {$obj delete}
}

# create a new view of a specific type on a given graph
proc newview {c g viewtype} {
    global debug

    canvas $c -bd 2 -relief raised -height 150 -width 150
    $c create text 35 10 -text $viewtype
    set v [dgview $viewtype]
    if {$debug} {foreach b [$v bind] {$v bind $b "+puts \"$b %v %n %e %A %a\\\""}}}

    # connect graph events to the view.
    # use the current position of mouse as the placement hint
    $g bind insert_node "+$c create \\\$shape \\\$boundary -tag %n -activefill yellow -fill white
        $v insertnode %n \\\$shape \\\$boundary -at \%x \%y
        $c configure -scrollregion \[%c bbox all\]"
    $g bind insert_edge "+$v insertedge %e
        $c configure -scrollregion \[%c bbox all\]"
    $g bind delete_node "+$v deletenode %n"
    $g bind delete_edge "+$v deleteedge %e"

    # connect view events to the canvas
    $v bind insert_node "+$c move %n %a"
    $v bind insert_edge "+$c create %A %a -tag %e -activefill yellow -arrow last"
    $v bind modify_node "+$c move %n %a"
    $v bind modify_edge "+$c coords %e %a"
    $v bind delete_node "+$c delete %n"
    $v bind delete_edge "+$c delete %e"

    # bind mouse events in this canvas to the event procs
    bind $c <1> "+adjust_xy $c %x %y; b1_down $c"
    bind $c <B1-Motion> "+adjust_xy $c %x %y; b1_move $c"
    bind $c <B1-ButtonRelease> "+adjust_xy $c %x %y; b1_up $c $g"
    bind $c <3> "+adjust_xy $c %x %y; b3_down $c"

    # return canvas for packing
    return $c
}

# create a graph
set g [dgnew digraph]
wm minsize . 20 20
if {$debug} {foreach b [$g bind] {$g bind $b "+puts \"$b %g %n %e %A %a\\\""}}}

# create one example of each view type
pack [newview .1 $g dynadag] -side left -expand true -fill both
pack [newview .2 $g geograph] -side left -expand true -fill both
pack [newview .3 $g orthogrid] -side left -expand true -fill both

```

6 Foundation Libraries

The layout algorithms underlying TclDG are implemented in C, and share a common external interface (ILE) and internal engine foundation library. We will briefly sketch the external interface. A layout engine offers the following primitives to clients:

- open and close a view
- insert, modify, and delete items in a view
- issue pending changes to a view
- return contents of views or attributes of items in views

At the time a new view is opened, the client passes a discipline structure that lists client-side callback functions for setting placement of objects in views, as well as the desired minimum separation and precision for placing layout objects, and a flag indicating if callbacks should be issued immediately when objects are moved (otherwise only issues on demand by the client.) The open call returns a handle to a new view that is bound to the given engine and discipline. Thereafter a client may edit the contents of a view, and receive callbacks corresponding to layout updates, and eventually close the view to release its resources.

The engine interface is not tied specifically to TclDG, so engines can be reused in other environments. For example a colleague wrote an OLE-compliant graph editor in Microsoft Visual C++ with this same collection of layout engines.

Layout engines are written on top of a library that supplies a common foundation for maintaining graph models of views, managing callbacks, fitting and clipping edge splines, and primitive geometric operations.

As previously described, TclDG relies on Libgraph for graph representation and the engines we have written also use Libgraph internally. Libgraph has about 60 interface functions. The main operations are

- open and close graphs and subgraphs
- read and write graph files
- search or traverse node or edge sets
- edit node or edge sets
- define and update string attributes

- bind runtime storage records to graphs, node, edges

Beyond basic node and edge set maintenance, Libgraph offers richer, more flexible semantic for more convenient programming. For example Libgraph has client-defined “disciplines” to customize memory allocation, I/O, object ID allocation, and callback management. This flexibility is particularly useful in TclDG. It supports general (not just node-induced) nested subgraphs, and a general purpose graph file language. Internally, Libgraph uses Phong Vo’s Libdict¹ to store node and edge sets. This provides efficient amortized log-time random access set operations using splay trees. To simplify efficient implementation of graph algorithms, certain key data access operations can be performed by in-line code. For example, node and edge sets in “read-only” mode may also be linearized or flattened into lists for traversal by in-line pointer-chasing code, and there is a similar way of binding run-time storage records for in-line access. For further background information, see the 1993 USENIX paper on earlier editions of Libdict and Libgraph [NV93].

Regarding layout algorithm efficiency, the engines we have written are advanced engineering test-beds, so we have not spent much effort on efficiency tuning. Our view is that the main problem at this stage is simply to understand what animation sequences of dynamic graphs look best, before trying to optimize their computation. DynaDag is usable on graphs up to several dozen nodes depending somewhat on the specific layout. An update always takes at linear time in the viewed graph size (because all object coordinates are recomputed); in the worst case adjusting a bad layout (such as one having many long edges or many edge crossings) could incur quadratic cost. Orthogrid employs a shortest path calculation on a set of tiles that is also potentially quadratic in graph size; though some performance enhancements borrowed from the DEC Contour tile-based VLSI router seem highly relevant [DM95].

7 Summary and Future Work

To recapitulate some of the virtues of TclDG’s design, as compared with conventional graph editor toolkits:

- Scriptable and customizable in Tcl.
- Extendible via other Tcl/Tk packages.

¹Recently extended to incorporate stacks and queues, and renamed LibCDT for C Data Types.

- Extendible via other programs using Libgraph's file language.
- Employs portable, efficient dynamic graph layout libraries written in C.

One of the main areas we have not addressed is efficient incremental layout when graph edits are compound, and perhaps large. An important example is batch loading of graphs from "save files" or externally-created diagrams. For acceptable performance, layout engines must load entire graphs more efficiently than by merely inserting individual nodes or edges and updating the entire diagram each time. They should also accept pre-existing coordinates as a strong hint about future placement. Our existing engine interface may be able to accommodate compound operations, such as batch loading, through the "immediate callback" flag. Compatible engines will need to record edits in a buffer, only process the edits and recompute layouts on demand. This implies some reworking of the engine foundation library.

Another idea is that the current framework of linked views in separate canvasses suggests literally embedding (nesting) diagrams inside other diagrams. In other words, this means implementing client requests to insert one view in another. Representing a nested diagram as a node (or some collection of proxy nodes and edges) with certain dynamic properties seems to be a natural approach, but new complications arise. For example, which engine is responsible for drawing edges that span nesting levels, and how does that engine obtain the edge's bounding path? Also, containers and containees potentially affect each other's layouts, so how should events be propagated between nested diagrams? Do nested layouts have fixpoints, or if not, how should convergence be enforced?

There is a good opportunity to improve the graphical presentation of dynamic layouts. Though the layout engines move objects atomically, smooth animation or fade-in/fade-out techniques are more understandable than simply switching frames instantaneously. On the other hand these graphical techniques also slow down dynamic graph displays so they may not be as appropriate for high volumes of graph data.

There is much interest in remote front ends written in Java [Ous95]. We are designing a prototype Java interface for graph diagrams. With this we hope to build distributed, multi-user graph applications in which a server maintains an abstract

graph and the clients present various views to the users and send user events back to the server.

We are writing a spring embedder layout engine for undirected graphs. Previous spring embedder layout programs have lacked efficient methods of resolving node-node, node-edge, and edge ordering constraints. Such constraints are vital to making layouts that are as readable as good hand-made diagrams. We are addressing these areas.

8 Release Plans and Tcl Compatibility

Our plan is to release TclDG within the next few months under the same license arrangements as graphviz.

TclDG is a loadable extension for tcl7.5 and tk4.1. TclDG is expected to be portable to all platforms on which Tcl runs, including Macs and PCs (to be verified, but all major parts have already been ported.)

TclDG requires DASH [Nij] and SPLINE patches (ours) to tk4.1.

References

- [Bou95] Thomas Boutell. gd1.2 - a graphics library for fast gif creation, 1995. <http://www.boutell.com/gd>.
- [Con95] Pad++ Consortium. Pad++ zoomable interface, 1995. <http://www.cs.unm.edu/pad++/>.
- [DM95] Jeremy Dion and Louis M. Monier. Con-tour: A tile-based gridless router. Technical Report 95/3, Digital Western Research Laboratory, 250 University Ave., Palo Alto CA 94301 USA, March 1995.
- [Dom95] Peter Domel. Webmap: A graphical hypertext navigation tool. *Computer Networks and ISDN Systems*, 28(1):85-97, 1995.
- [EN95] John Ellison and Stephen North. graphviz/tcldot release, 1995. <http://www.research.att.com/orgs/ssr/bookreuse>.
- [GKNV93] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE-TSE*, March 1993.
- [Him89] M. Himsolt. Graphed: An interactive graph editor. In *Proc. STACS 89*,

volume 349 of *Lecture Notes in Computer Science*, pages 532–533, Berlin, 1989. Springer-Verlag. <http://www.uni-passau.de/himsolt/GraphEd/graphed>.

- [MHT93] Kanth Miriyala, Scot W. Hornick, and Roberto Tamassia. An Incremental Approach to Aesthetic Graph Layout. In *Proc. Sixth International Workshop on Computer-Aided Software Engineering*, pages 297–308. IEEE Computer Society, July 1993.
- [Nij] Jan Nijtmans. dash patch. <ftp://ftp.nici.kun.nl/pub/tkpvm/tk4.1dash.patch.gz>.
- [Nor96] Stephen North. Incremental Layout in DynaDAG. In *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1996. <ftp://ftp.research.att.com/dist/drawdag/dynadag.ps.gz>.
- [NV93] Stephen C. North and Kiem-Phong Vo. Dictionary and graph libraries. In *USENIX Winter Conference*, pages 1–11, 1993.
- [Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*, chapter Simple Interactive Graph Editor, pages 213–215. Addison-Wesley, 1994.
- [Ous95] John Ousterhout. The relationship between tcltk and java, October 1995. <http://www.sunlabs.com/research/tcl/java.html>.
- [PT90] F. Newbery Paulish and W.F. Tichy. Edge: An extendible graph editor. *Software-Practice and Experience*, 20(S1):1/63–S1/88, 1990.
- [RDM⁺87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software-Practice and Experience*, 17(1):61–76, January 1987.
- [Tho95] Spencer W. Thomas. Tcl gd extension, 1995. <http://guraldi.hgp.med.umich.edu/gdtcl.html>.

Backtracking and Constraints in Tcl-BC

Dayton Clark and David M. Arnow
Dept. of Computer and Information Science
Brooklyn College
{dayton,arnow}@sci.brooklyn.cuny.edu

ABSTRACT: Tcl is extended to include backtracking controls and a facility for asserting constraints. This increases its flexibility in several domains and opens the possibility for its use in constraint-based logic programming applications and, conversely, applying constraint-based logic programming techniques to more traditional script language domains. The backtracking extensions are written entirely in Tcl itself, a fact which speaks well for Tcl's own extensibility.

1. Scripts and Controls

A good scripting language provides flexible control, data manipulation and convenient access to external facilities. The vast activity that Tcl [Ousterhout, 1990] has spawned speaks to its own merit in this regard [Tcl/Tk Workshops 1993, 1994, 1995]. Much of this activity has focused on extending the language, taking advantage of Tcl's elegant support for that endeavour. These extensions have primarily focused on extending Tcl's access to external resources. The most famous of these have brought Tcl to X [Ousterhout, 1990], to greater process and i/o control [Lehenbauer and Diekhans, 1992; Libes, 1990a, 1990b, 1991, 1993] and to TCP/IP [Smith et al., 1993a], and to audio and multimedia control [Jordan, 1994; Payne, 1993; Smith et al., 1993b; Duval, 1994]. A second group of efforts, most notably [incr Tcl], have sought to provide improved structuring facilities, primarily through object-oriented programming support [Braverman, 1993; Christopher, 1993; Howlett, 1994; McLennan, 1993 and 1994; Menges and Ladd, 1994]. Apart from this group, there have been few modifications of the language itself. To our knowledge, there have been no substantial extensions of Tcl's control structures, although Ousterhout explicitly invites practitioners to do so [Ousterhout, 1993].

In this paper, we present an *intension*^{*}, that is, an inward extension of Tcl, that derives a new control mechanism, backtracking, from Tcl's unique internal structure. We call this intension Tcl-BC[†] (Tcl with backtracking and constraints). We start by considering

at length the motivation for making backtracking available in the language, and then discuss the details of the intension in section 3, providing a few examples in section 4. In section 5, we describe our implementation and then close with a few thoughts about where our work will lead and some questions that it raises.

2. Wanted: Just One More Control Mechanism

Backtracking as a mathematical concept predates electronic computers by at least 60 years [Lucas, 1891] and is, of course, widely used as a programming technique in artificial intelligence and operations research [see, for example, Schalkoff, 1990]. The essential idea is that at some point in the computation a set of choices is encountered, only some of which may turn out to be viable. The choices usually involve the assignment of a value to a variable. The viability of the choices however can only be determined later in the computation. Backtracking allows the choices to be made, in some sequence, and with each choice, the computation proceeds, until "failure" (presumed lack of viability of the choice) is encountered. At that time, control returns to the *choice point*, some, *but not all*[‡], of the program state is restored, and the next choice in the sequence is made. When multiple choice points are involved, failure always returns to the most recent choice point encountered. If and when all the choices of a choice point have been exhausted, another failure is considered to have taken place, relative to the preceding choice point.

Backtracking has some of the flavor of `setjmp/longjmp`, but for a number of reasons, not the least of which is the fact that choice points may occur within procedures whose activation has ended, it involves a more complicated state restoration/preservation.

*. Our neologism, intended to express the idea that we are bringing forth a facility that was always implicit within Tcl. Thus, rather than extending Tcl outward to make external facilities available [as in, for example, Kenny et al., 1993; Richardson, 1993; Smyth, 1994; and Theobald, 1993], we are extending Tcl *inward* to make more available something that was, in a sense, always there.

†. That "BC" are the initials of our employer is purely coincidental.

‡. It is because *not* all the program state—including external side effects—is restored that computational progress is made possible.

Figure 1: Backtracking Through Lunch

```
choose (sandwich)
  cheese, pastrami, avocado
choose (sideorder)
  fries, yogurt, beef-barley soup
if (!kosher(sandwich, sideorder))
  fail
print sandwich, sideorder
fail
```

On the left is a sketch of a program that uses backtracking to enumerate all kosher* lunch combinations (within the confines of the rather limited menu!). The first, conditional, failure prevents non-kosher combinations from being printed. The failure at the end forces control back to a choice point in order that all combinations be found.

If the kosher predicate is computationally expensive, then in fact this would be a reasonable application of backtracking Tcl. Otherwise it is just a cute, but not very practical, example.

*. For the purpose of this example, we define kosher as simply not mixing meat and dairy products.

Although it is an elegant means of specifying a progression through a search tree, backtracking in itself does little to facilitate algorithmic efficiency. Search trees involving multiple choices tend to be combinatorially explosive. What is needed is a mechanism for prun-

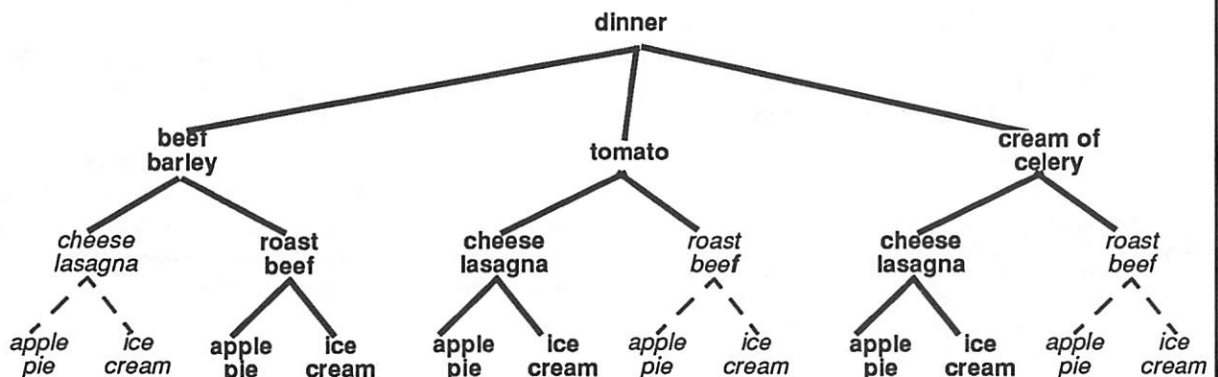
ing, that is generating failures *before* one reaches the leaves. One such mechanism is *constraints*: conditions that are applied before or during the search which, if violated at any time, cause immediate failure (i.e. return to the most recent choice point for the next choice).

Figure 2: Backtracking Through Dinner— With Constraints

```
constrain (not dairy(entree) and meat(soup))
constrain (not meat(entree) and dairy(soup))
choose (soup)
  beef-barley, tomato, cream of celery
choose (entree)
  cheese lasagna, roast beef
choose (dessert)
  ice cream, apple pie
if (!kosher(soup, entree, dessert))
  fail
print soup, entree, dessert
fail
```

In this example, the program asserts two constraints, which the backtracking mechanism will use to *prune* the search tree induced by the choice points. Only one failing leaf node (beef-barley, roast beef, and ice cream) is actually reached.

The pruned sub-trees are shown below in italicized, non-bold font with dashed edges. Note that these constraints reduce the number of leaves visited by a half.



Despite its utility, backtracking is usually not implemented at the language level, except in the Prolog family of languages. This is because:

- implementing backtracking using conditionals and recursion is a widely understood programming technique;
- backtracking requires an at least implicit distinction between memory objects that are restored to a prior state and those that are not—a distinction that does not come naturally in many languages;
- backtracking involves substantial interaction with the program memory state, complicating the semantics of the language and increasing run-time overhead and compiler complexity.

Backtracking makes great sense in a scripting language like Tcl, however. This is because:

- the purpose of a scripting language is often for fast prototyping or gluing other programs—

widely understood or not, script-writers are understandably not eager to re-invent the backtracking wheel;

- scripting languages tend to use implicit declarations—distinguishing between restorable and non-restorable memory objects can be just a matter of listing one set or the other;
- the use of scripting languages for fast prototyping or gluing other programs makes their own speed, and therefore the run-time cost of backtracking, less important than it would be in other contexts.

Finally, there are several characteristics of Tcl that especially invite an implementation of backtracking. First, is the well-known property of scripts as first class objects in Tcl and the availability of `eval`. Second is the curious fact that the Tcl run-time memory organization is not simply a global heap plus a stack of activation records. Consider, for example:

```
#!/usr/local/bin/tclsh

proc Mid {val track} {
    global inMid
    incr inMid
    if {$val > 0} {
        uplevel 1 Mid [incr val -1] [append track ":Mid"]
        Bot [append track ":Mid"]
    }
    incr inMid -1
}

proc Bot {track} {
    global inMid
    puts "Enter Bot -- # in Mid currently = $inMid"
    puts "  up 0 levels: level#=[info level ] ([info level 2])"
    puts "  up 1 level: level#=[uplevel 1 info level ]\
        ([uplevel 1 info level 1])"
    puts "  up 2 levels: level#=[uplevel 2 info level ] (Main)"
}

set inMid 0

Mid 3 "Calling-Trace:Main"
```

The output of this program is shown below. The calling-trace printouts (in the output) correspond to the traditional stack of activation records model and reflect the dynamic history of procedure invocation. The

actual environment in which `Bot` executes is given in the printouts of the levels. The next figure illustrates both perspectives when the first entry to `Bot` takes place.

```

Enter Bot -- # in Mid currently = 3
  up 0 levels: level#=2 (Bot Calling-Trace:Main:Mid:Mid:Mid:Mid)
  up 1 level: level#=1 (Mid 1 Calling-Trace:Main:Mid:Mid)
  up 2 levels: level#=0 (Main)
Enter Bot -- # in Mid currently = 2
  up 0 levels: level#=2 (Bot Calling-Trace:Main:Mid:Mid:Mid)
  up 1 level: level#=1 (Mid 2 Calling-Trace:Main:Mid)
  up 2 levels: level#=0 (Main)
Enter Bot -- # in Mid currently = 1
  up 0 levels: level#=2 (Bot Calling-Trace:Main:Mid:Mid)
  up 1 level: level#=1 (Mid 3 Calling-Trace:Main)
  up 2 levels: level#=0 (Main)

```

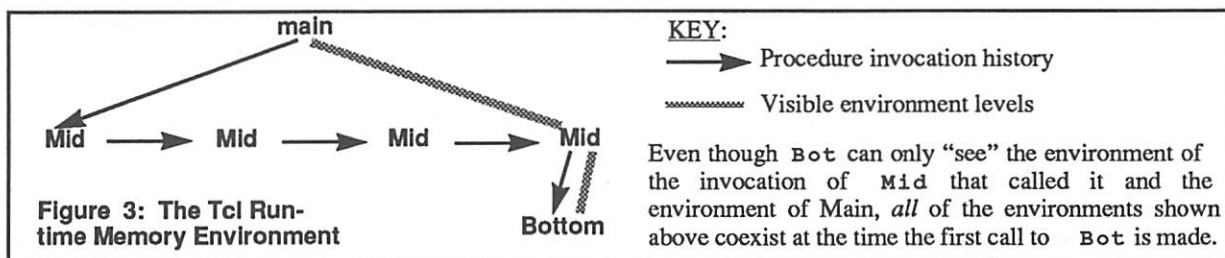


Figure 3: The Tcl Run-time Memory Environment

Thus there is implicitly a tree of environments in a Tcl computation. We will exploit this in our implementation, as described below in section 5.

2.1 Operations research problems

A substantial source of inspiration to us is the success of the language 2LP [McAloon and Tretkoff, 1995 and 1996]. 2LP combines imperative programming with logic programming facilities (backtracking and constraints) and a linear programming internal object. Compiled and highly efficient, 2LP focuses on a particular domain, and is therefore less general and flexible than C or Tcl. Among other applications, it has been used to implement a scheme for parallel integer goal programming applied to such classical operations research problems as job shop scheduling and the capacitated warehouse problem [Arnou et al. 1995]. The key to 2LP's success lie in these elements:

- an imperative programming model
- a logic programming control— in particular backtracking with constraints
- access to a linear programming solver
- articulation with C

The possibility of utilizing Tcl-BC in conjunction with a standard mathematical modeling package such as AMPL [Fourer et al., 1994], linked to a linear programming solver such as Cplex, is not far from our

thoughts. With the addition of access to tools like that, Tcl-BC could function as a highly extensible (as always) state of the art operations research system. Although Tcl's control could not compete in efficiency with 2LP, its advantages would lie in a greater ability to be linked with external resources (data filters, linear programming solvers, distributed programming extensions, etc.) and its greater programming flexibility.

2.2 Agents and backtracking with constraints

Another impetus for this work is the current interest in Tcl as a vehicle for (hopefully) intelligent knowledge agents roaming the internet [Johnson, 1993; Ott, 1994; Ousterhout, 1995]. Much of what internet agents will do is walk through what amount to search trees generated by choice points. Many of the edges in these trees will involve sending a Tcl script to a remote site for execution— a relatively time-consuming operation. Acting intelligently requires the ability to work within constraints, both prior (as in "don't buy anything that has less than a 1 year warranty") and dynamic (as in "don't investigate buying a system whose first two components are more expensive than an already known complete system"). It also requires the ability to *utilize* such constraints to minimize remote script execution.

Backtracking with constraints is a natural way to define such computations. Furthermore, if one views agents as Tcl scripts, it is likely that agents will be pro-

duced dynamically by some sort of GUI application. It is easier to automatically generate constraints and backtracking code to define an intelligent search than to try to build it into a more ad hoc mechanism.

2.3 Backtracking: why not just use Prolog?

Using Prolog entails a commitment to logic programming. The experience of 2LP demonstrates the convenience of providing a logic programming control within the context of an imperative programming model. That backtracking in Prolog turns out to be much faster than in Tcl-BC is no surprise and irrelevant—one doesn't choose a scripting language for the sake of speed.

2.4 Good and bad uses of Tcl-BC

The "hello, world" application of backtracking is the 8-queens problem, and we give, in the appendix, its solution in Tcl-BC. It is, however, precisely *not* the kind of application that Tcl-BC is intended for, except as an example of rapid prototyping. This is because the most of the computational effort lies in the making and unmaking of moves—and not in the computing the consequences of moves.

Good uses of Tcl-BC will be found wherever the cost of computing the consequence of a choice greatly exceeds the cost of making and unmaking choices or wherever significant constraints can be applied to prune the search tree, prior to or during the search. Thus, the use of Tcl-BC in constructing agents, where the consequence of a choice will typically involve the remote execution of a process, seems appropriate.

3. The Tcl-BC Extension

The backtracking and constraint extension to Tcl are implemented entirely in Tcl. It is found in a file called `bc.tcl`. Therefore no special installation procedures are necessary, simply `source` the extensions prior to their use.

The parts of the extension visible to the programmer fall into two categories: the commands which extend Tcl and implement backtracking and constraints, and the commands required because of implementation details which replace standard Tcl commands. The former include `bc_state` which specifies the state variables, `bc_fail` which induces backtracking, `bc_loop` and `bc_choose` which define choice points, and `bc_constrain` which specifies constraints. The latter include `bc_eval`, `bc_for`, `bc_foreach` and `bc_while` which substitute for the corresponding standard Tcl commands around code that includes choice points.

3.1 The backtracking calls: `bc_state`,

`bc_choose`, `bc_loop`, `bc_constrain`,
and `bc_fail`

The `bc_state` operation adds a list of variables (and/or arrays) to the *backtrackable set*, that is, the set of variables and arrays whose values are to be restored upon backtracking. Such variables and arrays are made global implicitly by this operation.

The `bc_fail` operation causes *failure* to take place, i.e. forces a backtrack to the most recent choice point, provided there is one. At that point, the values of the backtrackable set will be restored to that point and the next choice will be taken. If there are no more unexplored choice points, then all `bc_evals` (see below) fail and generates an exception.

There are two ways of establishing choice points: `bc_choose` and `bc_loop`. The form for `bc_loop` is:

```
bc_loop {initialization} {test} {step} {  
    body  
}
```

When a `bc_loop` is encountered, the *initialization* code is carried out and the *test* is evaluated. If the test fails (now or later), then the `bc_loop` fails, causing failure to take place. If the test succeeds, the *body* is executed once. If, during the execution of the *body* or *any code thereafter*, a failure takes place, the backtrackable set is restored to the state just prior to execution of the *body*, the *step* code is executed (generating the next choice), and the test is made again and the process repeats itself.

The form for `bc_choose` is:

```
bc_choose {choice1} ... {choiceN}
```

When a `bc_choose` is encountered, current choice is set to *choice1*. Only the current choice code (and not the other choices) is executed. If, during the execution of the current choice or thereafter, a failure takes place, then the backtrackable set is restored to the state just prior to execution of the current choice, and the succeeding choice becomes the current choice, providing there is one. If there is none, then `bc_choose` fails, causing failure.

The `bc_constrain` operation associates a condition with a list of variables or array-elements. After the operation, any change to that variable that causes the condition to be false generates an immediate failure. The form for `bc_constrain` is:

```
bc_constrain {variable list} "condition"
```

Note that if we write

```
bc_constrain {x} "\$x == \$y+1";  
# Backslashes needed to postpone $ evaluation.
```

a change in x's value that causes this condition to be false will generate a failure but a change in y's value that does the same will not! This is somewhat different from the usual logic-programming constraint.

3.2 The wrapper calls: `bc_eval`, `bc_for`, `bc_foreach`, and `bc_while`

Logically, the five backtracking operations described above are all that is needed and we truly wish we could end this section of the paper here. However, the backtracking extension requires the ability to gain control of the Tcl script itself in order to return to choice points. As a result, four additional calls are needed. They serve as wrappers of Tcl codes that contain backtracking operations.

First, a definition: *backtracking code* is any Tcl script that contains choice points (i.e. `bc_choose` or `bc_loop`) or that invokes a dynamic chain that includes activations containing choice points.

The `bc_eval` operation enables backtracking. Any backtracking code must be contained in an argument to `bc_eval`.

`bc_eval body`
where *body* is a Tcl script.

In addition, loops whose bodies contain backtracking code must replace `for`, `foreach` and `while` with `bc_for`, `bc_foreach` and `bc_while` respectively. Apart from enabling backtracking, the syntax

and semantics for these replacements are identical to those for the originals.

An important design consideration was that the extension leave as small a footprint on Tcl as possible. For this reason, the substitutes for the basic control commands is regrettable, but in each case required (Section 5 describes the implementation in some detail). Figure 4 summarizes the use of these wrappers. The substitutions are not syntactic sugar, the construct

```
bc_for {...} {...} {...} {
    code with choice points
}
```

is not a replacement for

```
for {...} {...} {...} {
    bc_eval {
        code with choice points
    }
}
```

The difference occurs if the program backtracks to a choice point within the loop. The `bc_for` loop retains control and its iterations will continue. The `for` loop does not regain control. After the `bc_eval` is completed execution continues with the command after the `for` command.

Figure 4: Backtracking and Non-backtracking code

```
source Bc.tcl

proc x {} {
    code with no choice points    Non-backtracking (needs no wrapper)
    bc_fail
}

proc y {} {
    bc_eval {
        code with choice points    Backtracking (needs wrapper)
        x
    }
}

bc_eval {
    bc_for {...} {...} {...} {    Backtracking (needs wrapper)
        code with choice points
    }
}

code with no choice points    Non-backtracking (needs no wrapper)

bc_eval {
    bc_while {...}{
        code with choice points    Backtracking (needs wrapper)
    }
}
```

3.3 Restriction

Variables used in the code that generates choice points in a `bc_loop` cannot be constrained. The example in the next section show how this restriction can be worked with.

4. Example

To demonstrate the look and feel of Tcl-BC, we consider an example of its use in solving the following famous cryptarithmic puzzle*:

```
SEND
+MORE
-----
MONEY
```

We will develop our program in steps. First, we source the essential `bc.tcl` and a useful script that defines the `wordexpr` procedure that we will use later.

```
#!/usr/local/bin/tclsh
source bc.tcl
source wordarithmic.tcl
```

The `cryptarithm` procedure will search for a solution; in doing so, it will create choice points, so its procedure body must be `abc_eval`. To represent the values of the letters appearing in the cryptarithmic puzzle, we will use the variables: `sendmory`. These will change value as the search proceeds, and will need to be restored upon failure and return to an earlier choice point and so they are arguments to `bc_state`. Three other auxiliary variables will be needed as well in the `backtracking set v vv ilet`. Their use will be explained below.

```
proc cryptarithm {} { bc_eval {
  bc_state v vv ilet
  bc_state sendmory
```

Next we set up an array `letters` that maps the integers 0 through 7 to the letters in the puzzle.

```
set nlet 0
foreach let {s e n m o r y d} {
  incr nlet
  set letters($nlet) $let
}
```

As arithmetically sophisticated human beings, we can give, in the form of constraints, the program some intuitions that are obvious to us. For example, we know that `m` must be 1, that `o` must be 0 or 1 and is therefore 0 and that `s` therefore is 8 or 9.

*. The puzzle is solved by finding an assignment of a decimal digit to each of the letters S,E,N,D,M,O,R,Y such that no digit is assigned to more than one letter and such that the above sum, with the letters replaced by the digits, is correct.

```
bc_constrain {s} "\$s>=8"
bc_constrain {e} "\$e<8"
bc_constrain {e} "\$e>1"
bc_constrain {n} "\$n==\$e+1"
bc_constrain {m} "\$m==1"
bc_constrain {o} "\$o==0"
bc_constrain {r} "\$r+\$n+1>=\$e+10"
bc_constrain {y} "\$y>1"
bc_constrain {d} "\$d>1"
```

For each of the letters in the puzzle (the `bc_for`), we must choose an appropriate value (the `bc_loop`). The former does *not* involve a set of choices (every letter in the puzzle must get a value) and so a `bc_for` suffices, but the latter does and therefore requires a `bc_loop` to generate choice points. The constraint that follows prevents the computation from assigning the same value to more than one letter. The constraint is written in terms of the value, `v`, rather than the particular letter `$letters($ilet)` in order to apply to all future values considered. A separate variable, `vv`, is used for generating the choices because of the rule that constraints cannot be applied to such a variable.

```
bc_for {set ilet 1} {$ilet <= $nlet
  {incr ilet} {
    bc_loop {set vv 0} {$vv <= 9}
      {incr vv} {
        set v $vv
        set $letters($ilet) $v
      }
    bc_constrain {v} "\$v != $v"
  }
}
```

It is only after the `bc_for` that we know that values have been assigned to all the letters in the puzzle. The call to `wordexpr` tests whether we have a successful combination. If not we fail, returning to a choice point generated by `bc_loop`. Otherwise, we print our solution and are done.

```
if {![wordexpr {
  send + more = money}] } {
  bc_fail
}
puts " \$s\$e\$n\$d"
puts "+\$m\$o\$r\$e"
puts "-----"
puts "\$m\$o\$n\$e\$y"
}
```

```
cryptarithm
```


The figure on the right shows the output of this program and its time on a Sparc10. When run with just one of the initial constraints,

```
bc_constrain {m} "\$m==1"
```

the program consumes about 50,000 seconds (14 hours) of Sparc10 time. When that constraint is removed... well, it is still running!

5. Implementation

There are three main aspects to adding the ability to backtrack with constraints to Tcl (or any language):

- the ability to maintain and restore the value of the state variables to their values at selected points (choice points) in the execution,
- the ability to ensure that constraints are not violated, and
- the ability to backtrack the execution itself to the choice points.

In Tcl-BC, the first two aspects are handled in a straight forward manner using the built-in Tcl facilities. The last aspect requires more code and is conceptually more involved, however, it is still done within Tcl itself.

Maintaining state. First, to maintain the state variables we keep a list of the variable names. Then when a choice point is encountered, we create a script which will restore the variables to their current values. The only non-trivial point is that we must make sure that state variables which don't exist when the choice point is encountered are made to not exist when we backtrack to the point.

Figure 5: cryptarithm on a Sparc10

```
iskra> time cryptarithm
9567
+1085
-----
10652
      124.2 real    58.6 user    0.0 sys
iskra>
```

Constraints. Tcl's ability to trap whenever a variable is modified via (the standard Tcl command) `trace` is the core of the implementation of constraints within Tcl-BC. Modification of any constrained variable invokes a command which finds the appropriate constraints, tests them, and causes a failure if the constraint is violated.

It is worth noting that the list of constraints is itself part of the state and is backtracked.

Backtracking. To backtrack the program flow requires the ability to continue execution at a choice point which may be anywhere within a script. The native Tcl `eval` takes an entire script as a single argument and provides no way to begin or continue execution at a point within the script. Hence, `bc_eval` was required.

`bc_eval` is meant to be functionally equivalent to `eval`. In fact, we imagine that future versions of Tcl-BC may "hijack" `eval`, replacing it with `bc_eval`.

The first thing `bc_eval` does is to turn its argument script(s) into a *strip*. The strip is a Tcl list of the first level commands in the script. Only the first level of commands are broken out, a command (such as `if` or `bc_eval`) which itself contains scripts becomes a single element in the list (see Figure 6, below).

Figure 6: Scripts and Strips

Script	Strip
<code>bc_state v vv ilet</code>	<code>"bc_state v vv ilet"</code>
<code>bc_state s e n d m o r y</code>	<code>"bc_state ..."</code>
<code>set nlet 0</code>	<code>"set nlet 0"</code>
<code>foreach let {s e n d m o r y} {</code>	<code>"foreach let ..."</code>
<code>incr nlet</code>	
<code>set letters(\$nlet) \$let</code>	
<code>}</code>	
<code>bc_constrain {s} "\\$s>=8"</code>	<code>"bc_constrain ..."</code>
<code>...</code>	<code>...</code>
<code>bc_for {set ilet 1} {\$ilet <= \$nlet} {incr ilet} {</code>	<code>"bc_for ..."</code>
<code>bc_loop {set v 0} ...</code>	
<code>...</code>	<code>...</code>
<code>}</code>	<code>}</code>
<code>if {![wordexpr ...] }</code>	<code>"if {![word ..."</code>
<code>...</code>	<code>...</code>
<code>}</code>	<code>}</code>
<code>...</code>	

The script (from the cryptarithm example) is transformed by `Bc_eval` to the list on the right, which is known as a *strip*. `Bc_eval` then evaluates each of the individual commands.

Since `bc_eval`'s will most frequently be inside of loops and evaluated many times, a list of encountered scripts and the corresponding strips is maintained. Subsequent invocations of `bc_eval` on a script then bypasses the parsing of the script.

Once the strip is available, `bc_eval` steps through the list evaluating each of the commands, applying `eval`. If one of the commands directly or indirectly invokes `bc_eval` then the current strip and the position within that strip is saved in a list. The new script is then converted to a strip and processing continues. When the end of the new strip is reached, processing of the interrupted strip is resumed.

This gives the flow of control within Tcl-BC an unorthodox flavor of machine language type control (complete with a program counter and branches) combined with implicit subroutine calls. In a sense, there are two simultaneous execution flows, the normal Tcl flow and the Tcl-BC flow. Consequently, naive combinations of `bc_eval` and normal Tcl may lead to unexpected behavior (see section 3.3, Restrictions).

Choice points are not explicitly entered by the programmer, they are implicit in the `bc_loop` and `bc_choose` commands. When a choice point is encountered at the current state (variables specified in `bc_state` statements) is saved in the form of a script that will restore the state. This script along with the current evaluation level (Tcl procedure nesting depth), a pointer to the current strip, and the current position within that strip are pushed onto a stack. Execution then proceeds.

When `bt_fail` is encountered, the choice point stack is popped, the evaluation level, strip, and position are restored, the script which restores the state is evaluated and execution proceeds from the restored position.

Currently, our primary difficulty with respect to compatibility with Tcl in general and other Tcl extensions, is the handling of returned values, from commands or procedures. Solving this problem is at the top of our agenda and it should be solved soon.

Typical presentations of backtracking invoke a tree structure to describe the flow of problem. Tcl-BC does not maintain an explicit tree but the list of partially evaluated strips can be viewed as a tree since each *thread* (as we call them) in the list points to a previous thread in the list. Imbedded in this list for a particular problem is the tree normally used to describe the problem.

6. Practical details: compatibility and availability

Compatibility. Tcl-BC is totally written in standard Tcl and in this regard should not interfere with any other extension. (Tcl-BC does reserve two prefixes, `bc_` and `_bc_`, for naming its procedures and variables). However, `bc_eval`'ed and `eval`'ed scripts cannot be freely intermingled (see section 3.3). Furthermore, `bc_eval` does not correctly implement returned values and procedure returns. These problems are the focus of our current work.

Availability. Backtracking Tcl is distributed as an under-1000 line script, along with documentation and examples. The relevant URL is

<http://acc6.its.brooklyn.cuny.edu/~arnow>.

Alternatively, send email to the authors.

7. Retrospect and Prospects

We view the development of Tcl-BC as double edged research. We are investigating the integration of backtracking into an imperative programming language and we are investigating the extensibility of scripting languages such as Tcl. We feel comfortable that our combination of backtracking and Tcl, while not seamless, shows that a smooth integration is feasible. As we stated at the outset of this paper, much of the work on extending the language Tcl itself has focused on adapting Tcl to be appropriate for writing very large Tcl programs. In contrast, our effort provides a powerful, expressive tool that makes it easier to use Tcl as a coordinating language.

False starts. As we discussed in section 2, the fact that Tcl can support a tree, rather than merely a stack, of activation records encouraged an implementation of backtracking. However, at the outset of this project we were also intrigued by the notion of using Tcl interpreters to represent active choice points. We were deterred from pursuing this by our inability to find a simple way for an interpreter to inherit state information, including position in a script, from another. Had there been a way to *fork* an interpreter, we could have eliminated the wrapper calls described in section 5.2, at the price of implementing the extension in C.

Regrets. We have already stated our concern over the need for wrappers. But these merely reflect a deeper, regrettable but apparently unavoidable aspect of the implementation-- its assembly code-like interpretation, with a program counter and all.

Why doesn't Tcl have.... Given the maturity of Tcl and our intuition that it was *right for this project*, we

worked for the most part on the assumption that whatever we needed *must* be available somewhere within Tcl. For the most part this was true, with the exception of the parser. It seemed surprising that, with Tcl scripts having the status of first class objects, there is no Tcl-provided mechanism for parsing them or at least a procedure to turn a script into a list of commands. The built-in command `info complete` performs the most tedious part of the parsing. We are also wondering whether a broader interpreter-creation/state inheritance mechanism would be possible. Besides facilitating an extension like this it might be very useful in threaded systems.

Future goals. We have several immediate goals. The foremost of these is to hijack Tcl's `eval`. This would eliminate the need for wrappers and, if done properly, reduce the likelihood of collisions with other extensions. Secondly, we would like to explore the performance impact of implementing this extension in C. In addition, we have plans to apply Tcl-BC to the applications cited in section 2.

8. References

- Arnold, D.M., McAloon, K. and Tretkoff, C.: Parallel integer goal programming. *Proceedings of the 23rd Annual ACM Computer Science Conference*. Nashville, Tennessee, March 1995.
- Braverman, M.S.: CASTE: A class system for Tcl. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).
- Christopher, W.A.: Writing Object-oriented Tcl-based Systems using Objectify. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).
- Duval, P. and Liao, T.: Tcl-Me, a Tcl Multimedia Extension. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).
- Fourer, R., Gay, D. and B. Kernighan, B.: *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, (1993).
- Howlett, G.: Packages: Adding Namespaces to Tcl. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).
- Johnson, R.W.: Autonomous Knowledge Agents - How Agents use the Tool Command Language. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).
- Jordan, E.M.: An Environment for the Development of Interactive Music and Audio Applications. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).
- Kenny, K.B., Sarachan, B.D., Sum, R.N., and Uejio, W.H.: Tcl/Tk - An Integration Vehicle for the Microwave/Millimeter-Wave Pilot Sites (MMPS). *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).
- Lehenbauer, K. and Diekhans, M.: Extended Tcl: Extended command set for Tcl, *unpublished manual page*, (January, 1992). This was cited by Lehenbauer in paper at one of the recent Tcl/Tk workshops. A more useful reference is: ftp://ftp.neosoft.com/pub/tcl/tclx-distrib/*
- Libes, D.: Curing Those Uncontrollable Fits of Interaction. *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, (June, 1990).
- Libes, D.: Using expect to Automate System Administration Tasks", Don Libes, *Proceedings of the 1990 USENIX Large Systems Administration Conference (LISA) IV*, Colorado Springs, CO, (October, 1990).
- Libes, D.: Expect: Scripts for Controlling Interactive Programs. *Computing Systems* 4,2 (1991).
- Libes, D.: Kibitz - Connecting Multiple Interactive Programs Together. *Software — Practice & Experience* 23, 5 (May, 1993).
- Lucas, E.: *Recreations Mathematiques* (1891).
- McAloon, K. and Tretkoff, C.: *Optimization and Computational Logic*, Series in Discrete Mathematics and Optimization, J. Wiley and Sons, to appear (1996).
- McAloon, K. and Tretkoff, C.: 2LP: Linear programming and logic programming, *Proceedings of Principles and Practice of Constraint Programming*, edited by V. Saraswat and P. VanHentenryck, MIT Press, 1995, pp 101-116.
- McLennan, M.J.: [incr tcl] - Object-Oriented Programming in TCL. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).
- McLennan, M.J.: [incr Tk]: Building Extensible Widgets with [incr Tcl]. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).
- Menges, J. and Ladd, B.: Tcl/C++ Binding Made Easy. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).
- Ott, M.: Jodler -- A Scripting Language for the Infobahn. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).
- Ousterhout, J.K.: TCL: An Embeddable Command Language. *Proceedings of the 1990 Winter USENIX Conference*, (January, 1990).
- Ousterhout, J.K.: An X11 Toolkit Based on the TCL Language. *Proceedings of the 1991 Winter USENIX Conference*. (January, 1991).
- Ousterhout, J.: *Tcl and the Tk Toolkit*, Addison-Wesley (1993).
- Ousterhout, J.: Scripts and Agents: The New Software High Ground. Invited talk at the 1995 Winter USENIX Conference, New Orleans, (January, 1995).

Payne, A.C.: Ak: An Audio Toolkit for Tcl/Tk. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Richardson, D.: Cooperating Applications through Tcl/Tk and DCE. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif., Berkeley, (June, 1993).

Schalkoff, R.J.: *Artificial Intelligence: An Engineering Approach*. McGraw-Hill (1990).

Smith, B.C., Rowe, L.A., and Yen, S.C.: Tcl Distributed Programming. *Proc of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Smith, B.C., Rowe, L.A., and Yen, S.C.: A Tcl/Tk Continuous Media Player. *Proc. of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June,

1993).

Smyth, D.E.: Tcl and Concurrent Object-Oriented Flight Software: Tcl on Mars. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Tcl/Tk Workshops: 1993, 1994 and 1995. The 3rd workshop was cosponsored by Usenix which will publish the proceedings. On-line proceedings of the 1st and 2nd workshops can be found at these ftp sites: <ftp.aud.alcatel.com/tcl/workshop/1993/tcl93-proceedings.tar.gz> and ftp.aud.alcatel.com/tcl/workshop/1994/1994_workshop.tar.gz

Theobald, D.: Interfacing an Object-Oriented Database System from Tcl. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

9. Appendix: 8-queens

The 8-queens problem is the standard illustration of backtracking in AI textbooks. The idea is to place 8 queens on the chessboard so that no queen is attacking any other queen. That amounts to choosing a set of 8 squares from among the 64 squares in the 8x8 chessboard such that no pair of squares lies on the same row, column or diagonal. The latter requirements are the *constraints* of the problem.

```
#!/usr/local/bin/tclsh
source bc.tcl
bc_eval {
    bc_state queen r c row column queen_list
    set queen_list {}
    set row 0
    set column 0
    set row_max 8
    set column_max 8
    bc_for {set queen 1} {$queen <= 8} {incr queen} {
        bc_loop {set r 1} {$r <= $row_max} {incr r} {
            set row $r
        }
        bc_loop {set c 1} {$c <= $column_max} {incr c} {
            set column $c
        }
        lappend queen_list [list $row $column]
        bc_constrain {row} "\$row != $row"
        bc_constrain {column} "\$column != $column"
        bc_constrain {column} "abs(\$column - $column) != abs(\$row - $row)"
    }
    puts stderr "Results--"
    foreach position $queen_list {
        puts stderr "    row [lindex $position 0] column [lindex $position 1]"
    }
}
```


QuaSR: A Large-Scale Automated, Distributed Testing Environment

Steven Grady, G. S. Madhusudan, and Marc Sugiyama
(steven@sybase.com, madhu@sybase.com, sugiyama@sybase.com)

Sybase, Inc.

Abstract

The *QuaSR* project at Sybase, Inc. involves the creation of thousands of automated tests for Sybase, Inc.'s *SQL Server*, each implemented as an independent Tcl program. The resulting test suite, significantly more than a million lines of code, comprises the largest known Tcl code base. The test harness is written in [incr Tcl] and the test cases in Tcl. Tcl and [incr Tcl]'s extensibility, simplicity, and reliability have made them uniquely suited to the development of a sophisticated automated testing system.

1. Introduction

QuaSR (Quality Systems Re-engineering, pronounced "quasar") is an internal project at Sybase, Inc. to improve the quality of its software, starting with its relational database server product, *SQL Server*. The primary technical component of this effort is the *QuaSR* test harness. The harness provides an environment in which small automated test programs may be developed to test arbitrary pieces of functionality, and sets of tests may be combined into single test runs, with distributed resource allocation being handled automatically.

This paper provides an overview of the design of the *QuaSR* test harness (also known as "QuaSR"), along with samples illuminating the use of the system. It concludes with lessons learned from the application of Tcl and [incr Tcl] to an automated testing environment such as ours.

2. Test Harness Design

2.1 Overview

The goal of the *QuaSR* project is to deliver a fast, reliable, extensible, and automated test system. Specifically, it must enable overnight execution of the main body of *SQL Server* regression tests. To this end, it provides for:

- minimized, automated resource allocation

- self-analyzing, assertion-based test cases
- distributed client/server execution
- standardized test components
- test case independence

2.2 Test Case Files

(To better understand the system, the following discussion will refer to the example Test Case File, `rtrim.tcf`. For the sake of brevity, the free-text sections have been edited.)

A *Test Case File*, or *TCF* (pronounced "tee-kiff"), is a file containing a set of test cases. The test cases are usually related in the functionality they test. The TCF consists of a declaration of resources available to all test cases, an initialization routine, called *tcf_start*, a termination routine, called *tcf_end*, and a set of one or more test cases.

In *rtrim.tcf*, one resource is declared, a standard database of type *stdbempty* (a database with a full complement of users defined, but no additional data). It is given the logical name *mydb*. In *tcf_start*, a database connection is established to the *SQL Server* on which *mydb* resides. The connection is established through the user *loginA* and given the name *sqlcon*. This connection is established before any test case in the TCF is executed. *Tcf_end* closes the *sqlcon* connection; it is invoked after all test cases are complete. The `tcf_run` line at the end is an imple-

```

resources {
    stdbempty mydb
}

tcf_start {
    mydb login loginA sqlcon
}

tcf_end {
    sqlcon close
}

testcase 1 -assertion {
    When "rtrim" is used with less than or greater than one argument, then
    error 174 is generated.
} -strategy {
    1. set up list of no argument and two arguments
    2. execute the following command:
        select rtrim(argumentlist)
    3. if SQL Server does not return error number 174 return FAIL
    4. repeat steps 2 and 3 for all test variations
    5. if all test variations are successful, return PASS,
        otherwise, generate UNRESOLVED
} -code {
    set pass_count 0
    set testlist {
        { }
        { abc, xyz }
    }
    set totvariation [llength $testlist]
    foreach test $testlist {
        set cmd "select rtrim($test)"
        SQL_cmd RESULT sqlcon $cmd
        if { ! [$RESULT servermsg 174] } {
            util_log "expected error 174 NOT returned by server"
            return FAIL
        }
        incr pass_count
    }

    if { $pass_count == $totvariation } {
        return PASS
    } else {
        error "Expected variations $totvariation but got $pass_count"
    }
}

tcf_run

```

rtrim.tcf

mentation artifact.

The *resources* section is simple and powerful. In *rtrim.tcf*, it contains only the declaration of a single database. However, QuaSR automatically supplies any additional resources necessary to support a standard database; specifically it results in the implicit declaration of a SQL Server to host the database, three logical devices to support the SQL Server and the database, three physical devices (potentially raw devices or filesystem space) to support the logical devices, a machine that holds the devices and runs SQL Server, and a network address for client-server communication.

Resources can also be declared more explicitly if non-standard configurations are required. For instance, a standard database could also be declared more explicitly as:

```
machine mymac
sql_server -machine mymac mysrv
stdbempty -sql_server mysrv mydb
```

The above declaration would ultimately result in exactly the same resource allocations as the simple declaration used in *rtrim.tcf*. The declaration syntax can be used to declare arbitrarily complex resource relationships, however. One can declare resources for tests that require interactions between multiple databases, SQL Servers, or machines, non-default device types and sizes, specific platforms or localization values, etc.

2.3 Test Cases

A *test case* is the smallest unit of functionality in QuaSR. Each test case is given an identifying number and has text describing the assertion under test, a strategy providing an English description of the approach used for the test, and a program implementing the test. The first two sections are normally logged when a test case fails so that all relevant information is readily available for test failure analysis.

rtrim.tcf contains one test case, which tests a negative assertion (one expected to generate an error). Using the resources declared in the *resources* section and the connection created in *tcf_start*, it creates the appropriate SQL commands to test the assertion, sends them through the connection, and analyzes the results. Depending on the results, the test case may generate a PASS, indicating the assertion was found to be valid, a FAIL, indicating the assertion was determined to be invalid, or an UNSOLVED (caused by the *error* call), indicating that there was some problem with the test case and that the

assertion could not be tested.

2.4 Test Sets

A test set is a set of test cases. An example test set is:

```
tests/dml/rtrim.tcf
tests/dml/select.tcf{1,3,5-8}
tests/ddl
```

This test set specifies a set consisting of all the test cases in *rtrim.tcf*, a subset of the cases in *select.tcf*, and all the test cases in the directory hierarchy *ddl*. Through other QuaSR mechanisms, it is possible to specify test sets such as “all tests that do not require a tape drive”, “tests that require *SQL Server* version 11 that run only on HPs”, or “tests that exercise code in *cache.c*”.

A test set is used as the basis for resource allocation and usually as the basis for an execution session.

2.5 Test Sessions

A test session normally consists of the following steps:

- generate
- scan_res
- acquire
- prepare
- exec
- release

The steps *generate* through *prepare* are used to set up the environment to run a set of tests. The *exec* step runs the tests. The *release* step cleans up the session, allowing any acquired resources to be used by others.

Generate is responsible for converting a test set specification file into a format appropriate for use by the rest of the system. For instance, it traverses any directory hierarchies to determine the individual TCFs within them.

Scan_res scans the resource requirements of the TCFs. It reads each TCF, determines the full resource requirements of each, then consolidates the full set of TCF-declared resources into a *minimal resource set*. Thus, for instance, a set of 100 TCFs may require only a single SQL Server, on a machine with enough space for two standard databases. (Theoretically, the resulting set may not actually be minimal, but in practice, our algorithm almost always generates a minimal set.)

Acquire uses the minimal resource set as a basis for requesting a machine (or machines) that can support the resources. The resources are allocated from the *Resource*

Manager, and are locked for use during the test session. By allocating the complete set of resources before execution, the user is guaranteed that the test run will not fail during execution due to lack of resources. Platform type, operating system, SQL Server version, and other attributes of the resulting execution environment are recorded for later phases.

Prepare prepares the resources. Support directories are created on the machines, SQL Servers are initialized, databases are created. This step is the first in which the machines responsible for executing the tests are actually used.

Exec executes the test cases. The user has the option of specifying an execution scenario that contains a subset of those in the original test set. Each TCF is run in turn, binding the logical resource names to the acquired resources, running the *tcf_start* initialization, executing each of the test cases specified in the scenario, then cleaning up in the *tcf_end* routine. The results are logged in a journal file, with summary information of the run printed on the standard output. The journal file is a structured file containing all information relevant to a particular run, including resource attributes, debugging information for non-PASSing tests, and timing information.

Release releases the acquired resources, freeing them for others to use.

The separation of the default process into individual steps allows for greater control by the user. For instance, a SQL Server developer can prepare all resources, run the tests, fix bugs demonstrated by the test run, then (using other steps not described above) copy a new version of the SQL Server binary to the acquired machines to re-run the tests. A test developer could modify the *resources* declaration in a TCF, then re-scan the resources to verify that the currently-acquired resources are sufficient to support the new declaration. There are also compound steps, such as "setup", which are responsible for executing multiple basic steps.

2.6 User Configuration

QuaSR allows the test runner to specify certain resource configuration values. For instance, the user may specify that the unspecified machine platforms should default to "Sparc" or that physical devices should default to use raw partitions. Another configuration variable allows the user to specify an alternative binary to execute in place of the standard SQL Server (it will be copied automatically to the remote machines before execution), or to al-

low the user to run binaries under a debugger rather than invoking them automatically.

2.7 Graphical User Interfaces

There are multiple graphical interfaces to the system. One provides an interface to the steps described above (along with information about the current state of the system, buttons to provide terminal connections to SQL Servers, and other conveniences). Another provides a simple way to browse a journal file. Among other things, it provides buttons to traverse the hierarchical format in an intuitive fashion, uses multiple colors to distinguish different types of information, formats query results into a familiar format, and allows the user to inspect the original TCFs.

2.8 Test Code Support

Various support libraries and extensions allow test case writers to write code at a high level conforming to that of the strategy. *QASQL* provides commands to communicate with *SQL Server*, formatting the resulting data stream into a manipulable format. *Undo* provides a mechanism for expressing an undo stack; this mechanism is used to record any changes that a test case makes, so that after the execution of the test case, the SQL Servers can be restored to the same state as before the test case was run.

The *Utility Library* provides a set of common high-level procedures and objects to simplify the coding of standard test steps (for instance, the *RESULT* object in the example test case understands requests for specific pieces of information, such as whether a given server message was contained in the result). The *Log* library provides a way to generate uniform, parsable debug messages suitable for later filtering.

The *Resources* module, which defines the resources available for declaration, supplies a variety of additional methods for information retrieval and resource control, such as determining the exact size of an allocated device, or shutting down and rebooting a SQL Server.

2.9 Utility Scripts

In addition to the journal browser, there are various tools available for analyzing the results of a single run or set of runs. There are tools to process the journal and place the results in a database suitable for querying. There are scripts to summarize the results of a single run and for comparing two runs. *Mrsummary* summarizes the resource requirements for a particular session. *Showsql* shows the SQL commands that were sent to SQL Servers

in a given session, optionally in a format suitable for input to *isql*, an interactive SQL shell.

2.10 Other Modules

The *agent* is a program that must run on each remote machine; it services requests to start processes and capture their output, create and remove files and directories, and other simple, operating-system-specific activities. It is the only part of the system which is ported to all the SQL Server platforms (which include, along with over a dozen UNIX variants, diverse platforms such as VMS, NT, OS/2, and NetWare). The *agentlib* library provides commands for the client to communicate with the server agents.

The *assertion database* stores information about assertions and their associated test cases. The *results database* stores the results of test runs (based on the contents of journal files).

Interactions in the system are controlled by a *state machine*, which determines what steps are legal at a given stage. This information enhances the main GUI, by limiting users to legal actions.

Tcl, [incr Tcl], and the various extensions are combined into a single interpreter, called *squash* (Sybase Quality Assurance SHell). *Squash* is the program which actually runs the code in the test cases.

There is an option to run a full test set in parallel. The algorithm used is very simple: to run n parallel threads, the tests are examined, and those requiring only a standard empty database are placed in $n-1$ homogeneous buckets; the rest in one heterogeneous bucket. The buckets are then separated into test runs and run individually. Currently, about two thirds of the TCFs fit into the homogeneous buckets. Ignoring time requirements for TCFs, on average a suite can be split into three threads to parallelize at maximum effectiveness.

3. SQL Server Test Suite

As of the time of writing (March 1996), thousands of individual test cases have been coded for testing SQL Server, comprising a Tcl code base with a line count in the millions – the largest known Tcl code base in a single project¹. Some of the cases test multiple variations; the total suite currently performs tens of thousands of product tests. Test suites for products other than SQL Server

are also in development.

The SQL Server test cases vary in length from around 10 lines to thousands of lines, with the majority of the simpler cases being under 100 lines. They vary in complexity from those that create and execute a single SQL command and verify the result (as in the test case in *rtrim.tcf*) to those that have several nested loops, and check the results of tens of SQL commands.

Individual test cases run in times ranging from under one second to about twenty minutes, with the vast majority running in under five seconds (depending on the server platform). On a fast server (e.g. an HP9000/800G with 128 megabytes of memory), a single-threaded run takes under twelve hours.

4. Use of Tcl and [incr Tcl]

The QuaSR test harness is implemented as a combination of C, C++, Perl, Bourne shell, Tcl, and [incr Tcl]. C and C++ are used primarily to provide Tcl extensions (including QASQL, the debug log library, and the agent library) that implement Tcl APIs on top of existing C APIs. Perl and sh are used for a few of the utility scripts, primarily performing file manipulation and process control. The bulk of the design is in Tcl (version 7.3) and [incr Tcl] (version 1.5). The GUIs are written using Tk (version 3.6).

4.1 Statistics

There are about 17,000 lines of Tcl/[incr Tcl] in the harness (plus about 8,000 lines more for the data that go into the standard databases), and 10,000 lines of C and C++. (There are also about 1,000 lines of Bourne shell and Perl scripts).

4.2 [incr Tcl] classes

The primary use of [incr Tcl] is in the definition of resources. Each resource is a separate class, organized into a single-inheritance hierarchy of about a dozen classes in total. Another hierarchy is used to define types of *Resource sets*, including the resources associated with a single TCF and the minimal resource set. Class containment is used to describe the relationship between platform-specific versions of SQL Server.

Other [incr Tcl] uses include: the utility library, which uses objects to create high-level interfaces to result streams coming back from SQL queries, and the resource manager, which defines machine attribute requests as objects.

1. This claim is based on responses to a query posted to comp.lang.tcl in February 1996.

4.3 Conceptual Expression

Nearly all of the conceptually challenging parts of QuaSR are written in Tcl/[incr Tcl]. The resource minimization algorithm, the dynamic resource binding, and the parallelization algorithm are coded in Tcl. The only uses of C/C++ are for C API module interfaces and for the journal browser. The browser must be able to parse and display files of 10 megabytes and more, so the processing code was rewritten in highly optimized C.

4.4 Test Case Tcl Usage

Because few of the test developers on the project had experience in object-oriented design and implementation, we decided that they could ramp up more quickly if they did not have to learn about [incr Tcl] and class design. In test case code, we minimize the exposure of [incr Tcl] to the use of method invocations on declared resources within the test cases. (Of course, we make full use of [incr Tcl]'s capabilities in the test harness code.)

5. Benefits of Tcl and [incr Tcl]

There was some initial apprehension about the choice of Tcl and [incr Tcl]. In hindsight, Tcl and [incr Tcl] have been suitable for both the harness and the test suite.

5.1 Easy to Learn

Part of the project was hiring and training a group of programmers who would develop tests under our system. At the time of hiring, almost none of them had used Tcl. Tcl's simplicity reduced the ramp-up time for developing tests under QuaSR.

5.2 Extensible

The use of [incr Tcl] classes as the basis for resource design made possible a truly powerful resource declaration language that can easily be extended as new resource requirements are identified. Even significant redesign of the class hierarchy has been possible due to the modular coding available with [incr Tcl]. During the course of the project, new resource classes have been added and old ones changed, with no backwards incompatibilities in the TCFs and few changes in other parts of the harness.

5.3 Embeddable

Standing alone, Tcl with [incr Tcl] would not have been sufficiently powerful to meet the goals of QuaSR in a reasonable timeframe. The embeddable nature of the Tcl interpreter made it possible to build a superstructure (*squash*) that defines procedures and objects at an appropriate conceptual level.

5.4 Interpreted

Because Tcl is interpreted, and because of the easy transformation between code and data, it has been possible to implement some very powerful constructs around the test cases. Test case code is stored as an [incr Tcl] instance variable in an object, and later executed. The execution is wrapped in code that catches exceptions, invokes user-specified hooks, and processes *undo* statements to reset the environment, all implemented with almost no effort. Adding further functionality, such as invariant checks between test cases, is similarly simple.

The interpreted nature of QuaSR has also made it easy to test the harness itself, since the tests have easy access to the internal harness procedures. Also, it was possible to implement simple tests for the GUIs by invoking widget actions via Tk's *send* command.

5.5 Public Domain

Because most of the components of QuaSR are based on public-domain code (Tcl, [incr Tcl], Don Libes' Tcl debugger, Perl, etc.), experimentation with different potential tools required little investment of time or money. The reduced risk made it possible to take more chances when trying to find the right solution.

The reliability of both Tcl and [incr Tcl] has been outstanding, attributable at least in part to their public-domain status, resulting in thousands of developers being able to identify and fix problems.

5.6 List and String Processing

Ultimately, testing SQL Server comes down to sending *Transact-SQL* commands to the server and verifying that the results are as expected. A language that allows strings to be created easily is appropriate to generating the commands to be sent, and a language that handles structured lists easily is appropriate for analyzing the response stream. The resulting test case code has been easy to read and write.

5.7 [incr Tcl] Cleanly Integrated

Minimizing the exposure of [incr Tcl] to the test case writers would not have been possible had [incr Tcl] not been cleanly designed. As it is, test coders do not need to learn anything about [incr Tcl] beyond the structure of a method invocation.

6. Disadvantages of Tcl and [incr Tcl]

While most of our initial concerns about the use of Tcl and [incr Tcl] were unfounded, there are a few areas that

need to be improved before their use in this project can be called a complete success.

6.1 Performance

Although the bulk of the processing during test case *execution* is on the server, much of the *preparation* time is on the client, particularly the *scan_res* phase which is responsible for consolidating the resources. The bottleneck here is [incr Tcl]'s performance when handling more than a handful of objects. [incr Tcl] 2.0 is supposed to alleviate this problem but we have not completed the transition to [incr Tcl] 2.0.

The time for simply parsing the TCFs is non-trivial when they comprise over a million lines, and complex algorithms can be painfully slow.

6.2 Memory Consumption

[incr Tcl] 1.5 is a memory hog. This becomes a problem when a large amount of data is held in memory. Again we are hoping [incr Tcl] 2.0 will alleviate this problem.

6.3 Lack of Development Tools

Although we make some use of public-domain development tools, such as Don Libes' debugger, we sorely feel the lack of industrial-strength tools,

- a truly integrated debugger that also understands [incr Tcl].
- a profiler with profiling for Tcl procedures, and [incr Tcl] objects and methods.
- a syntax checker.
- code coverage analyzer. In particular, each test case must be visually inspected to make sure there are no syntax errors, a tedious effort that a compiler would render unnecessary.
- Tcl compiler - we are testing a couple of recently released compilers.
- object browser for [incr Tcl]. This is mandatory if any serious programming has to be done in [incr Tcl]. Nautilus is a start, but we need tools similar to those supplied with most advanced C++ environments.

6.4 Rapid Change in Tcl Versions

QuaSR is an extremely large system which is rolled out to 300-500+ users currently. The frequency with which Tcl and its extensions get changed is a double edged sword. On one hand we get quick bug fixes, but the price we pay is frequent upgrades. Once the system goes production, frequent upgrades will not be feasible.

6.5 Error Reporting

Error reporting can be improved. The problem is mainly with the specificity of errors with respect to location and the nature of the problem.

7. Use of Alternate Test Harnesses

QuaSR uses the X/Open TET harness as its underlying harness. It was chosen primarily because of its successful use in other projects and its support for assertion based tests. Its distributed nature was also a factor in its selection. DejaGnu could also have been used but we were not sure about its maturity and ability to support distributed tests. Having said this, it should be pointed out that the test harness plays a very minor role in the current QuaSR system and hence the choice of a test harness is not very germane. This may change if the system is used for interactive testing, where DejaGnu has definite advantages. But in the current non-interactive regime of tests, most of the work is done by the test case and the distributed agent with the harness merely acting as a test case sequencer.

If we ever decide to switch to DejaGnu, the effort will not be significant since the underlying harness is fairly well isolated from the rest of the system.

8. Futures

QuaSR is undergoing continuing development. The following changes are expected in the near future:

8.1 New Features

QuaSR was originally designed for SQL Server testing. Relatively simple mechanisms can be added to support client testing as well, including interoperability testing of arbitrary client-server combinations.

Support for testing server products other than *SQL Server* can be added relatively easily through additions to the resource class tree. Once the new resources are in place, tests can simply declare the new resource and use it. The new tests would operate cleanly with tests in the existing suite.

8.2 Updated Software

QuaSR is currently being rolled out to various test and development groups at Sybase, Inc. In order to preserve stability, we have not integrated the latest versions of the underlying software. We expect to switch to Tcl 7.5, Tk 4.1, and [incr Tcl] 2.0 when we have the time to deal with any problems generated by the switch.

8.3 Performance Optimization

The current system meets the broad performance goals. However more work is required before all the specific performance goals are met.

9. Conclusions

The QuaSR project is ambitious both in its scope and its performance goals. The use of Tcl and [incr Tcl] as a basis for its implementation has its pros and cons but the benefits outweigh the disadvantages. The only significant drawback is the lack of industrial-strength development tools.

From a development standpoint, it is clear that the high-level nature of Tcl was beneficial. Both the harness design team and the test writing team found that the bulk of the design time was spent thinking about conceptual problems; once a solution was devised, it was straightforward to implement in Tcl. The developers spent their time much more effectively than had they been using C or another low-level language.

As a testing tool, Tcl's clean syntax and easy manipulation of structured data allow for powerful tests to be written simply and clearly. Inefficiencies due to interpretation are of little importance, particularly given the client-server nature of the product under test. Similarly, the high level interfaces made convenient by [incr Tcl] allow for powerful support libraries. High-level libraries allow tests to be written at a level close to that of their pseudo-code strategies.

Although there are challenges involved in using Tcl for a single, large-scale program, it is perfectly suited to the development of (thousands of) small programs. In particular, QuaSR, despite its size, has not been impaired by the use of Tcl, and in fact has been well-served by Tcl and [incr Tcl]'s reliability, simplicity, and extensibility.

10. Bibliography

- [Libe] Don Libes, "A Debugger for Tcl Applications", *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.
- [McLe93] M. J. McLennan, "[incr Tcl]: Object-Oriented Programming in Tcl", *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.
- [Savo] Rob Savoye, "The DejaGnu Testing Framework". Available at <http://www.cyg->

TclSolver: An Algebraic Constraint Manager for Tcl

Kevin B. Kenny
GE Corporate R&D Center
Schenectady, N. Y., USA
kennykb@crd.ge.com¹

Abstract.

TclSolver is a simple algebraic constraint manager for use in Tcl applications. It enforces two-way declarative relationships among variables. If, for example, it is provided with a formula like $A / B = C / D$, it can determine the value of any of the four variables A, B, C, and D when the user supplies values for the other three. It is not a general-purpose equation solver; it is limited to solving for variables that appear only once in their formulae. This paper presents the design and implementation of TclSolver, and an example application that uses TclSolver to perform simple engineering calculations on the World-Wide Web.

1. Introduction.

Constraint propagation is a technique that allows programmers to specify the relationships among the variables in a program declaratively, and update selected variables automatically in response to changes in other variables. This style of programming has been used for many years, beginning with Sutherland's seminal Sketchpad system [9]. It has been introduced into Tcl by the TclProp system of Iyengar and Konstan [3].

Constraint propagation is a convenient means for performing geometry management, for tracking state in a user interface, and for evaluating algebraic formulas. This paper concentrates on this last application, presenting TclSolver, an algebraic constraint manager that is used in a powerful spreadsheet-like calculator for engineering applications. TclSolver adds a new command, `solver`, to the Tcl language. The `solver` command allows the programmer to define a network of constraints (via the `load` and `compile` subcommands), to set and interrogate the values of variables (via the `set` subcommand), and to perform various housekeeping tasks. The `solver` command is implemented in C++,

and is, in fact, a Tcl-based interface to a C++ constraint system that can be used alone, without a Tcl interpreter.

The rest of this paper is organized as follows. Section 2 presents part of the history of declarative constraint management and introduces an engineering calculator application to show how constraint management can provide a powerful yet simple framework for calculations. Section 3 discusses the solver command, its use and its internal implementation. Section 4 presents experience with the solver command, including performance results, and Section 5 presents plans for future work in the area.

2. Background and Motivation.

The author first began working with constraint management in Tcl when work began on the ARPA CAMNET (Computer-Aided Manufacturing NETWORK) initiative [7]. Among the tasks undertaken was deploying four manufacturing applications on the World-Wide Web (WWW), one of which was an interactive calculator supporting the design of injection-molded plastic parts. An early prototype of the interactive calculator was built in Tcl and Fortran, using a locally-developed library of Tcl tools for WWW applications [2]. The prototype calculator was *unidirectional*; it took as input the dimensions of a part (a snap-fit finger) and computed several key attributes of part performance.

Users' initial reviews of this prototype were not altogether favorable. While the users were intrigued by the user interface, and particularly by WWW as the delivery vehicle, they had serious misgivings about the functionality. The commonest concern they raised was that the calculator had to be *bidirectional*. Sometimes, said the users, they wanted to compute the performance of the part from its dimensions. At other times, however, they wanted to specify the part's performance and compute dimensions that would yield that performance. Each of the fields in the worksheet describing the part ought to be both an input and an output. The system should calculate values that the user left blank based on the values that the user supplied.

1. This research was supported in part by the Advanced Research Projects Agency (ARPA) Manufacturing Automation and Design Environment (MADE) Initiative through ARPA Contract F33615-94-C-4400.

File Edit View Go Bookmarks Options Directory Window Help

Back Forward Home Reload Stop Open Print Find Stop

Location: <http://cobweb.crd.ge.com:80/cgi-bin/edc/edc/gear/4#CURRENT>

What's New What's Cool Handbook Net Search Net Directory Software

Spur Gear Calculations: (Contents...)

Tooth shape:

Number of teeth:

Lewis form factor $Y = 0.358$

D_p	Pitch diameter (in)	<input type="text" value="2.0"/>
F	Face width (in)	<input type="text" value="0.209497"/>
P_d	Diametral pitch (/in)	<input type="text" value="15.0"/>
P_c	Circular pitch (in)	<input type="text" value="0.20944"/>

- For safe torque limit for the life of the gear, enter one of the measures of service life, obtain the material's fatigue limit for the computed number of cycles, apply an appropriate safety factor, and enter the factored fatigue limit as bending stress.
- For maximum stall torque, enter the material's flexural yield strength as the bending stress.
- For stress analysis, enter applied force or torque and read the resulting stress.

NOTE: All calculations assume that a full width radius is extended to the tooth root. Plastic materials are notch sensitive, and shock loading may be a concern.

speed	Rotational speed (rpm)	<input type="text" value="360.0"/>
PLV	Pitch line velocity (ft / min)	<input type="text" value="188.496"/>
life	Service lifetime (hr)	<input type="text" value="10000.0"/>
cycles	Number of cycles in service life	<input type="text" value="2.16e+08"/>
T	Torque (in lbf)	<input type="text" value="12.0"/>
W_b	Tangential load (lbf)	<input type="text" value="12.0"/>
sigma_b	Bending stress at root (psi)	<input type="text" value="2400.0"/>

FIGURE 1. Worksheet from the engineering design calculator.

The value of PLV was obtained by computing

$$PLV = \pi * D_p / 12 * \text{speed}$$

The rule that computed it was titled:

Definition of pitch line velocity

FIGURE 2. Explanation of a calculation

Figure 1 shows a worksheet from the system as it was redesigned in light of these comments. This particular worksheet performs stress and strain calculations on gears [6]. The users also requested the ability to get explanations of how the system arrived at the values of the outputs — that is, what formulae were used to calculate them (Figure 2).

Now that the design was revised, there remained the question of how to implement it. Clearly, some sort of declarative programming was needed; coding all the combinations of inputs and outputs in a system of sixty or so equations would be a nightmare. It was also desirable to continue to integrate the system in a Tcl framework, because the CAMNET team already had a substantial body of software tools and expertise in integrating Tcl and the World-Wide Web. Fortunately, the author had seen the TclProp system (Figure 3) and realized that constraint programming in Tcl was viable.

TclProp itself, however, was rapidly rejected because it propagated values only in one direction. If it were used, each constraint equation would have to be entered many times. A typical equation like

$$y = \frac{Wx^2}{6EI} (3L - x)$$

would have to be solved by hand and entered five times, once for each variable appearing in the equation. This process would be tedious and prone to error. A more sophisticated system was needed.

The next route explored was to integrate a general-purpose equation solver like Mathematica, Macsyma, or Reduce. All of the commercial ones, however, had no way to divorce the solver itself from the user interface, making it infeasible to integrate them into the Web. The Bertrand augmented term rewriting system [5] looked promising at first, but its solution rules were too rudimentary to be useful, addressing only systems of linear equations. It appeared possible to implement a suitable constraint manager in less time than it would take to make Bertrand's rule base powerful enough to address the issues in question. Work therefore commenced on implementing TclSolver. To mitigate the risk of this approach, a parallel implementation of the calculator using a commercial numeric (as opposed to symbolic) constraint solver [10] was also pursued.

3. Implementation.

TclSolver comprises a single command, `solver`, that is implemented in C++, `lex`, and `yacc`. A use of TclSolver consists of three phases. First, a system of equations is created. Second, input values are supplied to the system, and the output values are calculated. Finally, the output values are queried, and, optionally, the equations that computed them are constructed.

3.1 Building a system of equations.

A system of equations is constructed by the `solver` command:

```
% solver mySystem
mySystem
```

It is then populated by either loading it from a file:

```
% mySystem load myFile
```

or compiling it from a Tcl string:

```
% mySystem compile {
    E : "Voltage";
    I : "Current";
    P : "Power";
    R : "Resistance";
    E = I * R           "Ohm's Law";
    P = I * E           "Power law";
}
```

In either case, the system comprises variable declarations, which are variable names followed by colons and titles of the variables, and equations (annotated with quoted titles). The equations are in a C-like notation. They support the operators `=`, `+`, `-`, `*`, `/`, and `**` (for exponentiation), and the functions `exp`, `log`, `sin`, `asin`, `cos`, `acos`, `tan`, `atan`, `sqrt`, and `square`. The `load` and `compile` subcommands may be invoked multiple times to add additional variables and equations.

Internally, a system of equations is represented as a network derived from its parse tree. The system of equations above, for instance, would be represented by the network in Figure 3.

3.2 Assigning and propagating values.

Once a system of constraints is built, the `set` subcommand is used to assign values to some set of its variables:

```
% MySystem set E 10.0
10.0
% MySystem set R 50.0
50.0
```

TclSolver operates by propagating values in the forward direction, as was pioneered in the “Constraints” system of Sussman and Steele [8]. In other words, it takes a set of known values and a network representing a system of equations. It examines each equation to find out whether the known values uniquely determine the value of another variable, and if so, defines that variable’s value as being known as well. To clarify this process, consider the constraint network for the system of equations in Section 3.1, which is shown in Figure 3. It comprises two multipliers, shown as triangles, and four cells to hold numeric values, shown as squares. Each multiplier allows passage of data in either direction; when values are known at any two of its ports, it asserts the value at the third.

When the value of E is set to 10, nothing further can happen (R , I and P are still unknown). Now the value of 50 is assigned to R . The multiplier at upper right has known values at two of its ports (E and R), so it can compute the value of the third (I) by dividing E by R . It does so, giving a value of 0.2. Now the multiplier at lower left has known values at two of its ports (E and I), and can multiply them to give the value of 2 for P . Similar data flows result from assignments to most other pairs of variables: E and R , I and R , P and I , or P and E .

All operators, not just multipliers, pass data in either direction. The number of required types of operators is therefore less than half of what one would expect at first. There is no need for a “divider” operator — a division is represented by connecting a multiplier backwards. The three equations:

$$\begin{aligned} A &= B * C, \\ B &= A / C, \text{ and} \\ C &= A / B, \end{aligned}$$

result in identical constraint networks. Similarly, subtraction is addition run backwards, and a single type of operator handles powers, logarithms and roots. Three operator types represent the three circular functions (and, of course, their inverses).

3.3 Getting outputs and explanations.

Of course, once the values of variables have been assigned and the constraints have been propagated, the program needs to be able to deal with the results. Computed values are extracted with the same `set` command:

```
% MySystem set I
0.2
```

The equations that derived a value can also be extracted, using the `source` command:

```
MySystem source I
{E / R} {Ohm's Law}
```

The `source` command returns a two-element list. The first element is the expression that was evaluated to extract the value, and the second is the title of the constraint that was used.

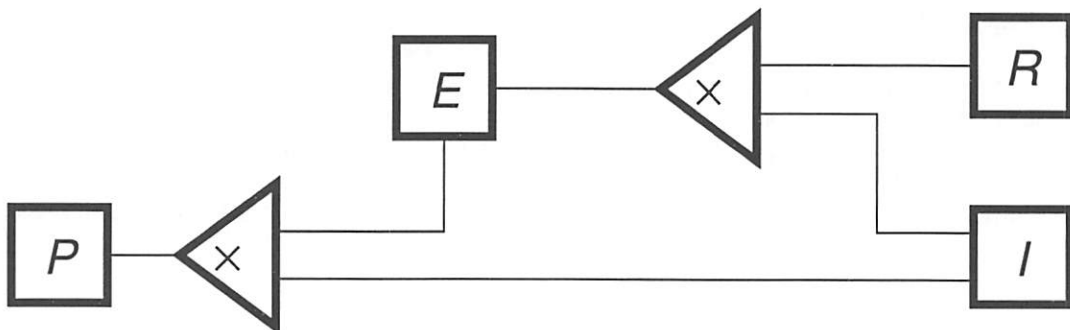


FIGURE 3. Sample constraint network.

3.4 Dealing with cycles in the constraint network.

TclSolver is not a general-purpose computer algebra system, even though its ability to propagate values through the constraint network in two directions gives something of the same flavor to it. In particular, there are common systems of equations that TclSolver is unable to solve. All of these involve cycles in the constraint network.

Consider, for example, presenting the network shown in Figure 3 with values for P and R . The constraint manager will be unable to determine E and I . The reason is that each depends on the other. Each of the multipliers will be unable to determine a value for E without knowing I , and vice versa.

This situation also arises in single equations. Any time that a single variable appears more than once in an equation, (for example, $y = (x - 1) / (x + 1)$), the resulting network will be unable to solve for that variable because it depends on itself.

It is possible to program around the difficulty by adding additional, redundant constraints. In Figure 3, for instance, adding the equation $P = I * 2 * R$ will allow TclSolver to solve for any two of the variables given the other two. The single equation $y = (x - 1) / (x + 1)$ can be solved by adding the redundant constraint $x = (1 + y) / (1 - y)$. The effort of programming these redundant constraints is less in TclSolver than it would be in a unidirectional system like TclProp [3] since only a few equations need to be inverted by hand.

3.5 Miscellaneous commands.

TclSolver has a few additional commands that query and maintain the state of the system. The `variables` command gets a list of the variables that have been defined. The `state` command determines the state (input, output, or unknown) of a variable. The `describe` command retrieves the title of a variable. These commands allow ugly first attempts at the user interface to be generated automatically and then modified by hand for improved ease of use.

3.6 Extending TclSolver.

Some calculations, obviously, cannot be expressed as simple systems of equations. TclSolver has been used successfully to combine complex calculations such as

numerical integration with simple relations. The complex calculations are integrated by augmenting TclSolver's C++ class library with new functions that perform the desired calculations. The calculations need not be invertible; the class library allows for operators that propagate data in only one direction. Typically, it takes anywhere from two to a few dozen lines of C++ code to add a new operator, depending on the complexity of its interface.

4. Experience with TclSolver.

TclSolver has been used to implement a plastics injection-molding calculator as an interactive World-Wide Web application. It has proven easy both to program and to use. Programmers report that the effort required is intermediate between setting up the same calculation in a spreadsheet and writing a program in a language such as Basic or C. Users report that the system is reasonably friendly, once they learn the paradigm. (Many reported confusion when first trying to adapt to the concept of a field that is both an input and an output.) A typical worksheet having sixteen variables, two dozen constraints, and graphs showing variation of two variables over time required 150 lines of Tcl code and 63 lines of source code for TclSolver, including whitespace and comments, and was implemented in an afternoon.

Performance results have been gratifying. Propagating the values through the constraint system takes a comparable amount of time to evaluating the same set of equations with the Tcl `expr` command. Attempting to manage forward evaluation only with Tcl `trace` operations roughly doubles the amount of time. TclProp [3] takes an order of magnitude more time when run in the forward direction only, and this time is increased by another order of magnitude if redundant constraints are added to allow values to propagate in both directions. Table 1 summarizes these results for the system of Figure 3. It reports the time to assign values to P and R , and extract the resulting values of E and I .

The chief drawback of TclSolver has been the limited capability of simple forward propagation as a solution technique. Programmers can easily miss cycles in the constraint networks, resulting in a worksheet's failure to solve for variables whose values are determined. Moreover, programmers and users have expressed the desire to solve in the backward direction, even when using unidirectional calculation techniques such as numerical integration. A few users have also asked for the ability to bound the calculations, generating error messages when values fall outside certain preset ranges.

TABLE 1. Performance of TclSolver relative to other techniques

Method	Time (ms)	
	Forward	Bidirectional
Tcl expr command	0.9	—
Tcl expr command inside trace action	1.8	—
TclProp	11.0	78.5
TclSolver	—	1.0

5. Plans for future work.

TclSolver has proven to be a useful tool in implementing spreadsheet-style calculations on the World-Wide Web. An initial release is available from the archive at <http://camnet.ge.com/software/>.

At present, TclSolver must be linked statically with a Tcl interpreter. It does not function as a dynamic load module, because the C++ code includes static objects that must be constructed, and the dynamic loader has no provision for invoking static constructor calls when a module is loaded. The team intends to address this problem in the future by replacing static objects with static pointers to objects that are constructed at run time.

The chief enhancements that are planned to TclSolver itself are inequality constraints and better handling of cycles in the constraint graph. Inequality constraints, initially, will simply report errors when the inputs fail to satisfy them, and will be used to bound calculations,

Special techniques will be required to handle cycles in the constraint graph. Given that certain of the functions will not be amenable to algebraic solution (several of the functions mentioned in Section 3.6 have no closed-form solution), it is anticipated that breaking cycles through algebraic manipulation will be infeasible. Rather, numeric techniques will be used, giving the TclSolver system something of the flavor of ThingLab [1] or Tk!Solver [4]. For variables that cannot be evaluated directly, the user will be prompted to supply guesses at the values, and a modified Newton-Raphson technique will be used to refine these guesses to consistent values. Optimization (minimization or maximization of a selected variable, subject to the constraints) will likely be added to the system at the same time.

6. References.

1. Borning, Alan. "The programming language aspects of ThingLab, a constraint-oriented simulation laboratory." *ACM Trans. on Programming Languages and Systems* 3: 4 (October, 1981) pp. 353-387.
2. Erkes, Joseph W., et al., "Implementing shared manufacturing services on the World-Wide Web." *Commun. ACM* 39:2(February, 1996) pp. 34-45.
3. Iyengar, Sunanda, and Joseph A. Konstan. "TclProp: A data-propagation formula manager for Tcl and Tk." *Proc. 1995 Tcl/Tk Workshop*, July 1995, Toronto, Ontario, Canada, pp. 25-30.
4. Konopasek, Milos, and Sundaresan Jayaraman. "Constraint and declarative languages for engineering applications: The Tk!Solver contribution." *Proc. IEEE* 73:12 (December, 1985) pp. 1791-1806.
5. Leler, Wm. *Constraint programming languages: Their specification and generation*. Reading, Mass.: Addison-Wesley, 1988.
6. Roark, Raymond J., and Warren C. Young. *Formulae for stress and strain*. New York: McGraw-Hill, 1975.
7. Sobolewski, Michael W. and Joseph W. Erkes. "CAMNET: Architecture and applications." *Proc. Concurrent Engineering 1995 Conf.*, McLean, Virginia, August 1995, pp. 627-634.
8. Sussman, Gerald J., and Guy L. Steele. "Constraints: A language for expressing almost-hierarchical descriptions." *Artificial Intelligence* 14:1 (January, 1980) pp. 1-39.
9. Sutherland, Ivan. "Sketchpad: A man-machine graphical communication system." Lincoln Laboratory Technical Report 296. Cambridge, Mass.: MIT, January, 1963. An abridged version appears in *Proc. 1963 AFIPS Summer Joint Computer Conf.*
10. Vanderplaats, G. N. "ADS: A Fortran program for automated design synthesis." Santa Barbara, Calif.: Engineering Design Optimization, 1987.

The NR Newsreader

Jonathan L. Herlocker
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
herlocke@cs.umn.edu

Abstract

NR is a point and click GUI interface for browsing Usenet news. The NR interface is built using the Tk interface toolkit, and coded entirely in the Tcl scripting language. NR was designed as a framework for pursuing research into electronic information browsing. As a result, NR had strong requirements for configurability, extensibility, portability, and performance. This paper describes how those requirements were met through the use of features and extensions of the Tcl/Tk scripting language. The paper also describes some of the information filtering technologies implemented in NR.

Background

Usenet news is a medium that provides for collaboration and sharing of knowledge on a global scale through a bulletin-board model. Usenet is organized into thousands of newsgroups, each of which represents a general topic of interest. Every message posted to a newsgroup is seen by all readers who subscribe to that newsgroup. As Usenet news has increased its readership to millions, the quantity of text posted daily to popular newsgroups exceeds the amount of information that a person has time to process. As a result, reading Usenet news can be a frustrating process. A user must scan hundreds of subject lines each week in order to locate interesting information.

In an attempt to lessen the workload of users reading electronic news, I am experimenting with agent technologies that help a user locate information of interest. One of these technologies is content-based: an agent that observes and remembers the content of user-selected articles and then attempts to suggest new articles that contain similar content. The other technology, part of the GroupLens[1][7] project, is collaborative based. GroupLens helps a user to locate new information based on the opinions of other users with similar interest. Both of these technologies are described in more detail at the end of the paper.

After designing the architecture of my agents, I found that there were no existing newsreaders that met my requirements for a prototype system. I wanted a GUI-

based newsreader to provide more flexibility in creating interfaces for visual feedback. Existing GUI-based newsreaders such as XRN[6] did not lend themselves to easy extension of the graphical interface.

I chose to implement a prototype newsreader in Tcl/Tk, primarily because I knew from previous experience that I would be able to quickly build a working interface. In addition, the Tk text widget had the desirable property that it both understood how to lay out text efficiently and supported embedded graphics and controls.

After working part time for three months, I had a working prototype that supported my agent assistant. However, I was unable to do a suitable experiment to determine the usefulness of my agent. Users were unwilling to spend their time using a newsreader that did not support the features that they were used to. Therefore I decided to refine my prototype into a full-featured newsreading application that not only supported the most popular newsreading features, but was highly extensible, allowing it to be a testbed for agents such as the ones I designed. Having a liking for simplistic terms, I called the newsreader *NR*. Figure 1 shows a screen shot of NR browsing the newsgroup comp.lang.tcl.

Requirements

There were four major requirements that would make NR possible and successful.

1. Ease of coding
2. Configurability
3. Good performance
4. Portability

Ease of Coding

NR was not a project that I could afford to spend all my time on, so developing it required that it take minimal effort of coding. Using Tcl helped to satisfy that requirement. Tcl shortened the development time considerably over that of a compiled language such as C.

Tcl is an ideal language for a Usenet client, because both the data and communication protocols in Usenet are entirely textual, and require little support for binary datatypes.

One of my constant paranoidias is that I will reinvent the wheel every time I write a piece of code. However, I was much more at ease when writing NR, due to the enormous amounts of code that I reused in developing my application. Of the 16,360 lines of Tcl code currently in the NR application, only 6,350 lines were written by me.

The majority of the code I reused came from the exmh mail reader[10]. Organizing code in fashion similar to exmh (as described in [10]) is a good way to create modularization for a language that provides little built-in support for it.

However, one of the frustrations that I experienced when writing NR was that there was no good place that I could go to locate code modules for code reuse. Current Tcl/Tk archives provide a centralized location for Tcl/Tk source code and applications, but these archives focus on applications and not on reusable code modules. In addition, all the sites provide the applications and

code modules in a form of a monolithic list with nothing more than a one or two line abstract. A much more structured interface is needed for a Tcl/Tk code modules archive. This ideal archive for Tcl/Tk code modules will be hierarchically organized, so that if a programmer wants to find a common interface element (such as an extended listbox, file selection box, or extended canvas widget), he can move directly to a point in the hierarchy and only browse through applicable modules. The abstract text of all code modules and applications should be searchable, so that users can locate applications of interest, and programmers can locate Tcl code modules if they are unsure of its location in the hierarchy.

The high-level language features of Tcl made the coding and debugging process easy. Memory management bugs, which are my primary source of errors in other applications were non-existent in Tcl. Tcl's use of a single string data-type freed me from much of the work of designing data-structures and made passing data to other

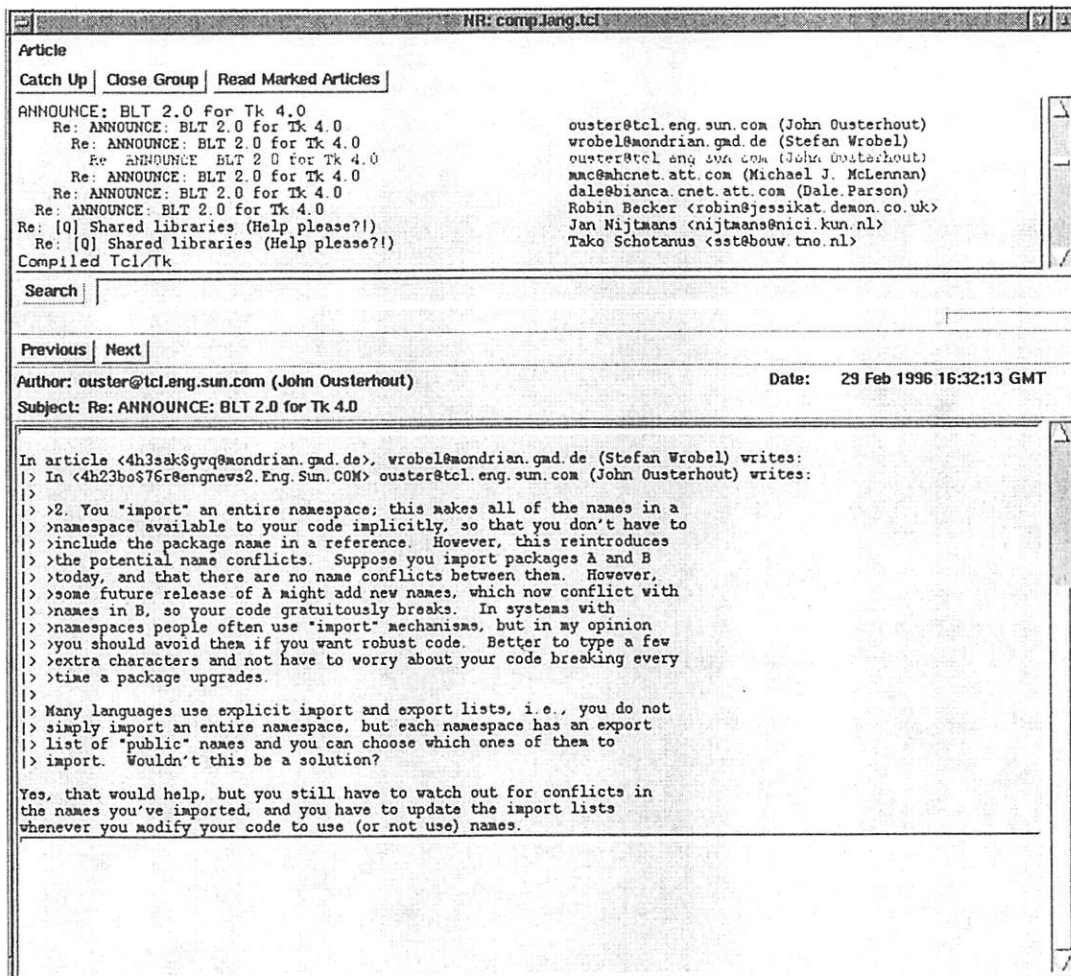


Figure 1: Snapshot of the NR group view window while reading the newsgroup comp.lang.tcl.

people's code modules simple.

Tcl-Prop[2] was a Tcl extension that I found simplified a good portion of the interface code. Some of the most monotonous busy work in programming modeless interfaces such as Tk is the enabling and disabling of controls based on their applicability to the current state of the application. The standard way to handle these controls is to find every location in your code where the state of the program changes and insert a call to the enabling/disabling routine. However, the state can change in many locations in your code, possibly even in somebody's else's code. Another method is to use Tcl-Prop, a data-propagation formula manager. Tcl-prop is a Tcl-only extension which provides a runtime constraint system in some ways similar to that found in Garnet[5]. Tcl-Prop will automatically change the state of the controls whenever the state of the application changes. Setup of the constraint system is simple. You call one procedure to define how the state is encoded, then provide a Tcl script to be executed whenever the state changes. State can be as simple as a variable, or something more complex like window visibility. Tcl-Prop allows you to trigger Tcl scripts on events that occur in code modules without editing the code of those code modules.

Configurability

Every user of Usenet news has developed their own hab-

its of browsing electronic news, and many have no desire to change their habits. This was my stumbling block when I sought to obtain useful data to document my agent assistant. So NR had to support commonly desired features and be easily extended to support other features that users might desire. I also wanted to design NR such that it could be easily extended for research involving information browsing such as user trace collection or development of agent assistants. Overall, NR had to be an application that was highly configurable in terms of features and interface.

I chose to incorporate much of the configurability code from the exmh mail reader. The configurability module in exmh provides for customization through the use of options and X-defaults. Users can not only set application variables, but specify new button and menu widgets to be added to the interface through the use of X-defaults. The exmh preferences module also takes a list of available preference options and builds a GUI interface for manipulation of those options. A personal Tcl library allows users to override parts of the exmh code with their own personalized code.

However the model of editing X-defaults and adding Tcl code can be too complex for many casual users, especially if they are not literate in Tcl. These casual users may often wish to take advantage of new features that have been added by others or configurations created by

```
set moduleName "killFromAOL"
set moduleVersion "1.0"
set moduleDescription "This module will cause NR to discard all posts
    from authors at AOL (*@aol.com)"
set moduleHelp "By default, the killFromAOL module will alert the user
    before discarding any posts, but killFromAOL can be configured to
    quietly and indiscriminantly kill all posts from AOL. See the
    killFromAOL preferences for more."
Preferences_Add "Kill AOL" \
    "These options control preferences for the Kill AOL module" {
        killAOL(showStats) \
        killAOLShowStats \
        ON \
        { Show statistics } \
        "This setting controls whether or not a dialog box will
        inform whenever AOL articles are killed, how many articles
        were killed and provide you with the option of overriding."
    }
}
NR_AddKillPattern author {*@aol.com}
```

Figure 2: Example feature module that will cause NR to discard all articles originating from authors at aol.com.

others. To serve their end, NR supports "One-file feature modules." A user who extends the NR interface can share that extension with other users by simply mailing them a single Tcl file. A simple GUI interface allows novice users to install new feature modules easily, as well as inventory, enable, disable, and remove feature modules. In addition, NR supports versioning of modules, to prevent conflicts among multiple versions of the same module.

For example, consider a feature module that causes your newsreader to automatically discard all postings from authors at a specific site. The feature module might look something like the one shown in Figure 2.

A feature module contains four parts: the name and version of the module, a description of the module and help text on how to use the module, a list of user-configurable preferences for the module, and the code to implement the module. A description of the preferences code can be found in [10].

Adding new features to NR is made easy by providing a documented set of hook functions, which allow users to specify Tcl scripts to be executed at key points of the application. Statistics on newsreading can be collected through hooks that are called whenever an article or group is read. Extensions can control how and what articles are displayed through hook functions. Together these hooks can provide information and control to intelligent agent assistants.

Good Performance

In order to create a viable user interface to Usenet news, NR has to provide comparable performance to other existing newsreaders which may be written in compiled languages such as C. Speed was a considerable problem with the initial implementation of NR. There were three sources for poor performance: the network protocol, content parsing of text, and insertion of text into the text widget.

Since electronic news is a data intensive medium, the network and NNTP[3] news server were the primary bottleneck. NR attempts to lessen the impact of the delay by using incremental loading. NR incrementally loads newsgroups upon start-up, allowing the user to start reading the first newsgroup as soon as it becomes available, while continuing to download other newsgroups in the background.

Due to the inherent performance limitations of Tcl, post-processing of article text proved that it could be time consuming. In several cases, NR needs to parse large text articles, looking for regular expressions or patterns. Currently there is no good solution for these situations,

however in the future, NR may prefetch and preprocess article text off-line. Prefetching will also hide the network delay.

Inserting large text files into text widgets results in delays which cannot be fixed by preprocessing, and cannot be canceled. The text widget could better handle large text insertions by occasionally processing idle events. This would allow users to page through the early parts of the text while the later parts were still being loaded in.

Portability

In order to reach a large cross section of users on the internet, NR had to be portable to a variety of architectures. Tcl was an ideal language, because the Tcl language is almost entirely architecture and operating system independent. However there were some barriers to portability.

NR requires network socket code to communicate with the NNTP server. Until recently, NR required the C-code extension Tcl-DP[9] to provide these socket commands. A special wish had to be compiled with the Tcl-DP extensions in order to run NR. With the release of Tcl7.5alpha, NR was distributed with a loadable Tcl-DP extension that was dependent on the binary architecture. Now that Tcl7.5beta has support for native sockets, NR will be a Tcl application that requires no extensions to the core Tcl interpreter.

The cross-platform support introduced in Tcl7.5 and Tk4.1 has made NR accessible to millions of Microsoft Windows users on the Internet. However making NR runnable on a Windows machine required considerable modification, due to heavy use of the "exec" Tcl command in some of NR's code modules. Here are some suggestions to writing portable Tcl code:

Avoid exec whenever possible. This one should be clear. Programs that exist on Unix may not exist on Windows and vice versa.

If you must use exec, don't hard-code the commands into each exec. Use a variable to represent each command, so that you can change your code to use a different command by only changing a line or two. `exec $rmProg $filename` is much better than `exec rm $filename`. On a similar note, don't use `/bin/sh` to start your commands or rely on shell specific substitutions or commands. This mistake has caused me considerable grief.

Remember that the format of X-defaults is completely foreign to Windows users. There is nothing wrong with using X-defaults to save your preferences, but don't expect Windows users to edit those files directly.

Resource Locking in Tcl

NR maintains a single network socket for communication with the server. Since NR allows the user to be browsing multiple groups at once (each in a separate window), access to the network socket must be controlled to prevent multiple groups from accessing the socket simultaneously. Some method of resource locking must be used. Locking is not completely intuitive in Tcl, because only a single execution context is supported. However, there turns out there is a very simple solution to resource locking in Tcl. A single global variable is used to represent the lock and processes desiring to obtain the lock enter a busy-wait loop, using the "after" command. For example, any Tcl procedure needing to access the network socket would have a form similar to the following:

```
Proc GetOverview { group } {
    global socketLock
    if { $socketLock == 1 }
    {
        after 500 [info level 0]
        return
    } else {
        set socketLock 1

        # Main code of procedure
        # follows
        ...

        set socketLock 0
    }
}
```

Upon failing to acquire the lock, the procedure uses the after command to schedule itself again in 500ms, in hope that the lock will then be free. [info level 0] returns the current procedure name and arguments. This simple form of resource locking works because Tcl has

only a single thread of execution and reads and writes to the lock variable are guaranteed to be atomic. Those more familiar with the theory of process synchronization will point out that the method described above is not guaranteed to be fair. Processes waiting for a resource are not guaranteed to be serviced in a first come-first served manner. This was not a requirement in NR.

Examples of Using NR for Information Filtering

GroupLens is a collaborative-based filtering system that is currently being targeted on Usenet news. A GroupLens-enhanced newsreader provides predictions as to how interesting a user will find each news article. GroupLens works by having users assign ratings to each news article they read based on how interesting they found that article. Ratings are sent to a central GroupLens server, known as the Better Bit Bureau or BBB. The BBB correlates all the users' ratings and clusters users of similar interests. The BBB then generates predictions for each user based on the ratings of users in the same interest cluster. The newsreader client can then choose how it wishes to use the predictions. It could just display them (as shown in figure 3), sort them, or even choose to display only articles with a prediction above a certain threshold.

Another filtering agent that I am working on is completely content based. This filtering agent requires no explicit interaction from the user, gathering all its information by simply watching the keystrokes and mouse events of the user. When reading news, a user selects potential articles from a list of all available subject lines. The filtering agent works on the assumption that there is some method to the manner in which a user chooses certain subject lines and ignores others. The agent remembers those subject lines chosen for reading as positive examples, also remembers lines that were ignored as negative examples. The agent builds an AI structure

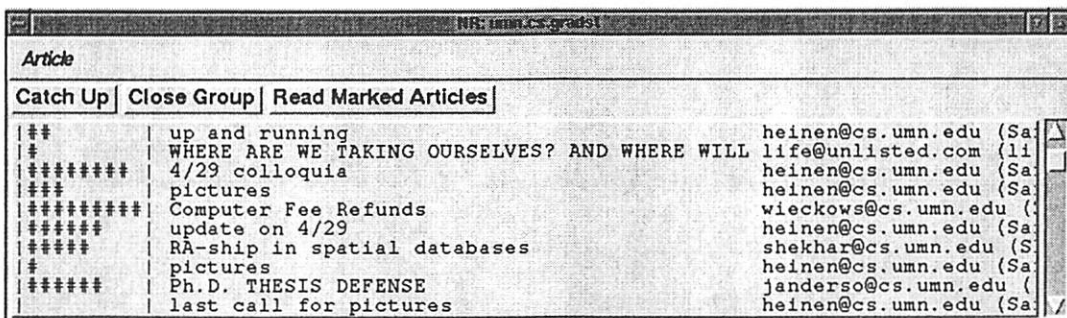


Figure 3: NR with GroupLens predictions. The predictions are represented here as ASCII scales. The larger the scale, the better the prediction.

called a decision tree, based on what keywords signify interesting articles and what keywords signify bad articles. The decision tree can then be used to classify new articles as interesting or uninteresting. Figure 4 shows NR displaying ratings on a A-F scale. Since the displayed ratings are actually menubuttons embedded in the widget, a user can click the mouse on a rating to determine exactly why that subject was assigned a certain prediction as shown in Figure 5. The resulting menu also offers an option to provide feedback as the accuracy of the prediction. This feedback can then be used to help improve the building of the decision tree.

Current Status of NR

An alpha release of NR is currently available from `ftp://ftp.cs.umn.edu/dept/users/herlocke/nr`. The extensibility interface and documentation have not been completed yet. Once work on both these issues is completed, a beta test of NR will be announced. If you wish to be added to the announcement mailing list, please send me a email note (`herlocke@cs.umn.edu`).

Conclusions

NR has grown from a simple prototype to a large application of around 16,000 lines. The process of evolving

NR has been relatively painless because the Tcl language supports and encourages extensibility. I have shared some of my key experiences from developing NR, and I hope that the Tcl/Tk community will continue to develop highly configurable and extensible applications. To restate the key points:

Tcl/Tk is an excellent language for rapid development of full featured applications. It's not just for creating prototypes. Performance problems can usually be solved through proper structuring of the application.

Organize your code in a consistent, modular fashion. Follow a code organization scheme such as the one described in [10], or perhaps use the object-oriented facilities described in [incr Tcl][4], OTcl[8], or Object Tcl[11].

Share your useful code. If you have written a useful code module, don't be afraid to share it with the rest of the Tcl/Tk community. Chances are that one of us would like to use it too. But please document it well.

Write easily extensible applications. Ideally, application extensions will be exchanged among users casually, promoting all users to share their labors of customization.

A runtime constraint system is an excellent way to trig-

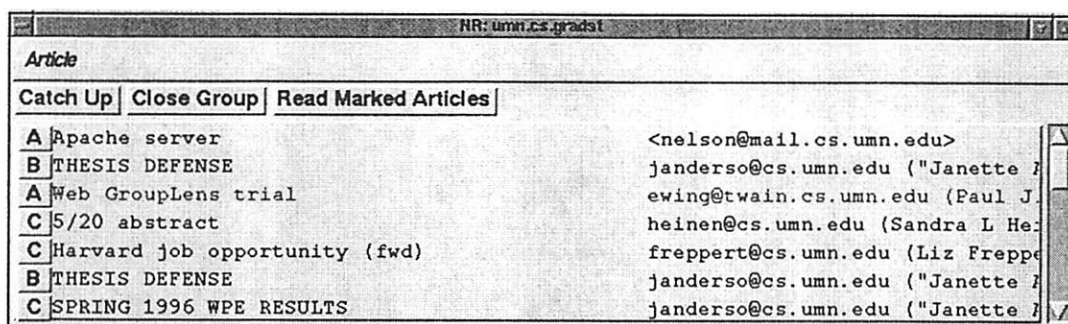


Figure 4: NR with A-F predictions, based on keywords found in the subject line. Note that the ratings are actually menubuttons embedded into a text widget.

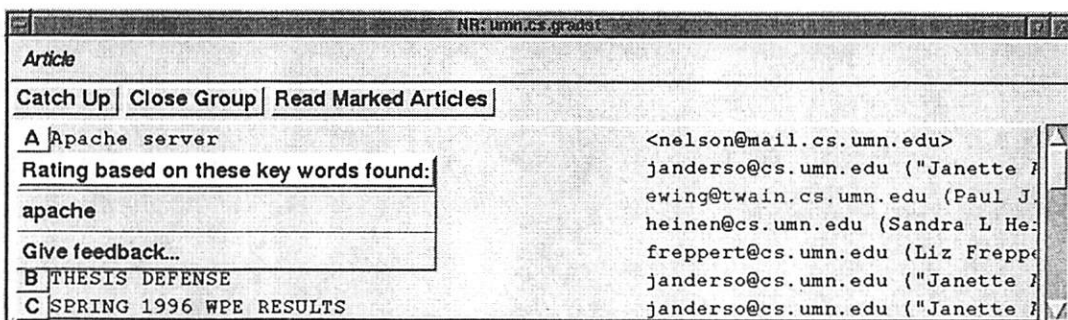


Figure 5: The embedded menubuttons are used both to provide more information about a prediction, but also allow the user the opportunity to provide feedback on a specific prediction.

ger events based on state changes. Tcl-Prop works wonders.

Don't compromise portability with use of exec. Try and find another way.

Appendix - Features of NR

- Intuitive GUI interface based on Tk
- True multilevel threading based on "References:" lines.
- Support for display of MIME articles.
- Optimized for use over slow network links such as 14400 modems.
- Support for browsing multiple newsgroups simultaneously
- Highly customizable interface.
- Large shared code base with the popular exmh mail reader allows sharing of code extensions

References

1. GroupLens. <http://www.cs.umn.edu/Research/GroupLens/>
2. Sunanda Iyengar and Joseph A. Konstan. TclProp: A Data-Propagation Formula Manager for Tcl and Tk. *Proc. of the 1995 Tcl/Tk Workshop*, Toronto, Ontario, Canada. July 1995.
3. Brian Kantor and Phil Lapsley. Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News. RFC 977. <ftp://ds.internic.net/rfc/rfc977.txt>. February 1986.
4. Michael McLennan. The New [incr Tcl]: Objects, Mega-Widgets, Namespaces, and More. *Proc. of the 1995 Tcl/Tk Workshop*, Toronto, Ontario, Canada. July 1995.
5. Brad A. Myers, et. al. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment," *IEEE Computer*, November 1990.
6. Ellen Santovich, Rick Spickelmier, and Jonathan Kamens. The XRN newsreader. <ftp://ftp.cam.ov.com/pub/xrn>.
7. Paul Resnick et. al. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. *Proc. of the ACM 1994 Conference on Computer Supported Cooperative Work*. Chapel Hill, NC. 1994.
8. Dean Sheehan. Interpreted C++, Object Oriented Tcl, What next? *Proc. of the 1995 Tcl/Tk Workshop*, Toronto, Ontario, Canada. July 1995.
9. Brian C. Smith, Lawrence A. Rowe, Stephen C. Yen. Tcl Distributed Programming, *Proc. of the 1993 Tcl/Tk Workshop*, Berkeley, CA, June 1993.
10. Brent Welch. Customization and Flexibility in the exmh Mail User Interface. *Proc. of the 1995 Tcl/Tk Workshop*, Toronto, Ontario, Canada. July 1995.
11. David Wetherall and Christopher J. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. *Proc. of the 1995 Tcl/Tk Workshop*, Toronto, Ontario, Canada. July 1995.

Tcl/Tk in the Development of User-Extensible Graphical User Interfaces

John M. Skinner*, Richard S. LaBarca[‡], Robert M. Sweet*

**Biology Department, Brookhaven National Laboratory,
Upton, NY 11973 skinner@bnl.gov, sweet@bnl.gov*

[‡]Carnegie Mellon University, Pittsburgh, PA 15213 rlabarca@cmu.edu

Abstract

This paper describes the use of Tcl/Tk in the development of *user-extensible* Graphical User Interfaces. We describe a GUI that provides two facilities, employing Tcl/Tk, that allow its users to extend the basic GUI at runtime. The first facility is a built-in mini Tcl/Tk GUI builder that gives users the ability to generate new screens that are automatically incorporated into the existing GUI. This is accomplished at runtime and does not require the user to write any code. A second feature allows for the incorporation, control, and access of data from, pre-existing Tcl/Tk screens without modification to the original source code of those screens. Program extensions accomplished with these facilities are integrated seamlessly into the original program, and are available immediately, as well as in subsequent executions of the program. In addition to the demonstration of a specific application of these methods, we will show that the techniques are transferable to other applications.

Background

The main subject of this paper is a GUI that is used to run instrument-control programs [1] at Brookhaven National Laboratory's (BNL) National Synchrotron Light Source (NSLS). The NSLS is a world-class research facility used by over 2500 researchers each year to perform experiments in physics, chemistry, biology, materials science, and various other technologies. The instrument-control programs originally had text-based, command-line driven interfaces. Since there are only a handful of facilities such as ours in the world, and many scientists are able to use the available research time, users are under significant pressure to make the best use of the limited time they are given. The result is that they are often sleep-deprived, stressed, and prone to making errors. In response to this situation, we have invested a significant effort in the development of user-friendly software for data acquisition and analysis.

The main GUI for our instrument control programs

was written in C with the aid of the Builder Xccessory GUI builder [2]. Developing the GUI presented an interesting challenge in that the underlying programs provide a C-like macro language that allow users to define new functions at runtime. Users frequently develop customized macros for these programs at numerous experimental stations. A significant percentage of a user's interaction with these programs is through these new macros, which are typically different at each station using the software. Since it was impossible to anticipate what interfaces for future program extensions should look like, or to construct interfaces for all of the important macros already written, we realized that users would need to be able to extend the interface themselves. Clearly, we could not assume that users of the program would have the programming abilities or the time necessary to modify the code of the basic GUI. In addition, since the macros can be defined and incorporated at runtime, we felt that the interface extensions should also be able to be developed and applied without exiting the program.

It seemed clear that if an end-user were to be able to extend the GUI without writing any code, that some sort of "GUI builder" would need to be provided. Although Tcl/Tk builders such as SpecTcl [3] are excellent tools for program developers, someone using these builders would still need to have some knowledge of Tcl/Tk to write the callback procedures. In addition, we felt that the comprehensiveness of a tool such as SpecTcl required some initial investment of time on the part of the user in order to learn how to use the builder properly. We therefore decided to write our own built-in, mini GUI builder. This would enable us to ensure that the code generated by this tool had the capability for two-way communications with the existing program without requiring modification. We also could write our builder so that no coding was necessary to add new screens.

Design Decisions

Our main objective in designing our builder was ease-of-use. We wanted the users to be able to

generate useful interfaces without having to spend time thinking about how to use the builder. This led to trade-offs. Clearly, the more features we included, the more complicated the builder would be to use and, we felt, the less people would be willing to use it. We decided to limit ourselves to the generation of labels, entries, and pushbuttons. Although the resulting screens would contain only this limited set of GUI components, these would be sufficient to produce useful interfaces that provide distinct advantages over command-line execution. When using one of the generated screens, the user will immediately know information about the program command, such as its name, the number of arguments, something about those arguments from their labels, and reasonable default values. In addition, the screens can display important results or status during and after execution of the command, and the user may easily re-execute with minimal or no editing.

Although other components such as toggle buttons and radioboxes would enhance the appearance of the resulting screens, it was not felt that the potential for enhancement outweighed the corresponding increase in the complexity of using the system. We also bore in mind that we planned to supply a facility, described later, that would accommodate those wishing to invest the time to create more elaborate interfaces.

Generating New Screens

Figures 1 through 3 illustrate the steps involved in generating a new screen with our "GuiBuilder". Pushing the "GuiBuilder" button in the "Beamline Macros" menu will cause a form to pop up which will request the name of the command to call, the number of inputs, and the number of outputs (Figure 1). Based on this information, a final form will be displayed that will request label names and default values (Figure 2). Applying this information will result in the generated screen being displayed (Figure 3). It will contain the requested input and output fields, default values, and "Go" and "Close" buttons. Pushing the "Go" button will execute the command with the supplied input. Results can be displayed in the output fields. In addition, a button will be added automatically to the Beamline Macros menu (Figure 3) so that this screen may be called up again. The fact that this command will now be accessible from a pulldown menu is in itself a significant advantage. The menu of extensions serves to remind the user of the most important commands in the program and only requires that they recognize a command that they need to run, as opposed to having to remember its

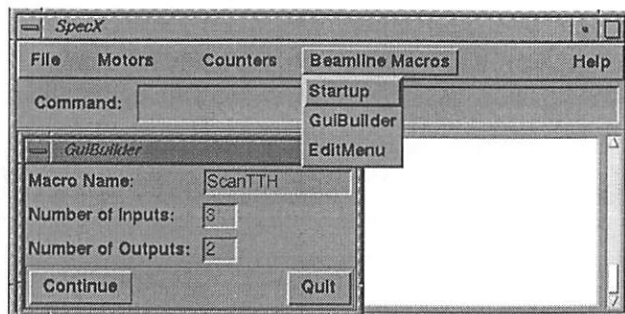


Figure 1. GuiBuilder: step 1

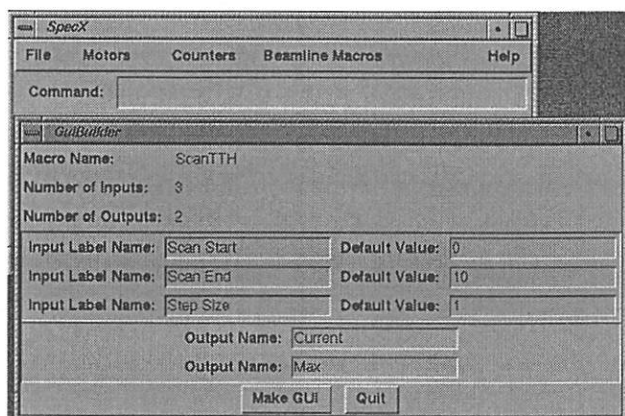


Figure 2. GuiBuilder: step 2

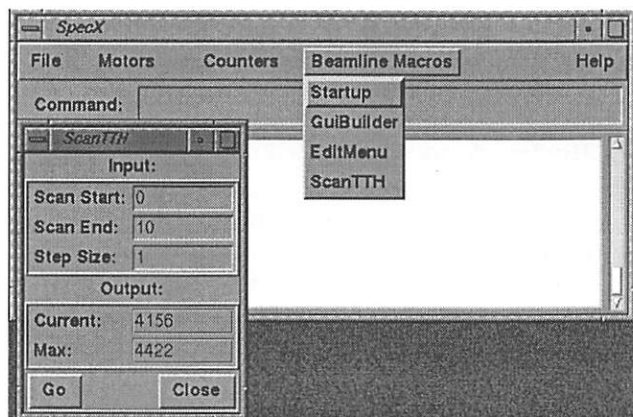


Figure 3. GuiBuilder: result

name, or be forced to query the program for all available commands.

The following is a listing of the macro that the new screen was generated for:

```
def ScanTTH '
#s1=start,$2=end,$3=step size
maxint = 0
for (cp=$1; cp<=$2; cp+= $3) {
```

```

mva tth cp
# next line puts intensity in val
read_intensity

# the next line causes the
# intensity to appear in the
# "Current" field of the generated
# screen
sendgui "ScanTTH" "Current" val

if (val > maxint) {
    maxint = val
}

# display the maximum intensity in
# the GUI
sendgui "ScanTTH" "Max" maxint
}

```

The GuiBuilder produces different results based on the user's entering a zero in the number of input and/or output fields requested when interacting with the first form. If only output fields are requested, then the resulting screen will be configured as an output-only status screen. These screens will not contain input fields, default values, or a go button. They are intended to make important status information easily visible to the user.

If a user specifies zero inputs and zero outputs in the first form, then no second form will be displayed, and the result will be a command button added to the "Beamline Macros" menu.

Incorporating User-Written Screens

To accommodate experienced Tcl/Tk programmers who wish to develop more complex screens and integrate them into the GUI, we have provided a second extension facility, called "EditMenu". This aspect of the program is used to incorporate user-written Tcl/Tk screens that have no special provisions for communications with other processes. After being added to the existing GUI with EditMenu, the Tcl/Tk variable values of the newly incorporated screens may be accessed from the instrument-control program by calling "guival", and set with "sendgui". More detailed control of a Tcl/Tk screen can be accomplished by using the "gui_exec" call from the underlying program. This function allows one to send Tcl/Tk code segments to a screen. This can be used for such actions as modifying values, changing colors, adding new GUI components, and binding new actions. One may actually use "gui_exec" to perform all of the functions that are accomplished with

"sendgui" and "guival". When a screen is added with EditMenu, it will then become an item in the Beamline Macros menu. We also supply a "popgui" command that will provide us with the same communication mechanisms as EditMenu, but will not result in an addition to the Beamline Macros menu.

The access and control of the screens incorporated with EditMenu or "popgui" is accomplished without editing the Tcl/Tk source. Our method does not require any particular X server authorization scheme, and is done in a way that does not compromise the security of the system. One may also edit the Tcl/Tk scripts so that they issue commands to the program from which they are controlled. This is done by calling "send_command" from inside the Tcl script. "send_command" will send any string, typically a command, to the underlying program. It can be called by any script that has been incorporated with EditMenu or popgui. It is also possible to use "gui_exec" from inside a macro to bind a send_command call to a widget instead of editing the script. Figure 4 details the profiles and usage of the calls that interface the underlying program with the Tcl/Tk scripts.

The "SpeedControl" window in figure 5 is a Tcl/Tk GUI that was constructed with SUN's SpecTcl GUI Builder. This was done independently of the existing GUI. That is, there is no special code included to facilitate communications with other procedures or

Tcl/Tk GUI Communications Command Summary
<p><i>sendgui</i> - Send output to a Tcl/Tk GUI. usage: sendgui <screen name><variable name><value> example: sendgui "speed control" "speed" speed_variable</p>
<p><i>guival</i> - Retrieve a value from a Tcl/Tk GUI. usage: guival <screen name> <variable name> the_value = input() #need to read the result example: guival "Controlit" "var1" var1_value = input()</p>
<p><i>gui_exec</i> - Send Tcl/Tk code to a screen to be executed. usage: gui_exec "screen name" code example: gui_exec "SpeedCtl" "bind .quit <Button-1> exit"</p>
<p><i>popgui</i> - Pop up a user-defined GUI. usage: popgui <screen name> example: popgui "temperatures"</p>
<p><i>send_command</i> - Send a command from Tcl/Tk to the underlying program. usage: send_command <some_command_string> example: button .go -text "Go" -command\ {send_command drive_motor}</p>

Figure 4. Communications commands summary table

processes and the buttons have no command bindings. The bindings are added by the macro listed below for the purpose of demonstrating the `gui_exec` call.

```
def wire_speed_buttons '
    code = sprintf("bind .go
<Button-1> {send_command
RunMotor}")
    gui_exec "SpeedControl" code
    code = sprintf("bind .quit
<Button-1> exit")
    gui_exec "SpeedControl" code
```

By using the EditMenu facility (Figure 5), the screen is merged into the existing GUI, and will be used to control the speed of some imaginary device. For demonstration purposes, this device will be another Tcl/Tk screen containing some scale and entry

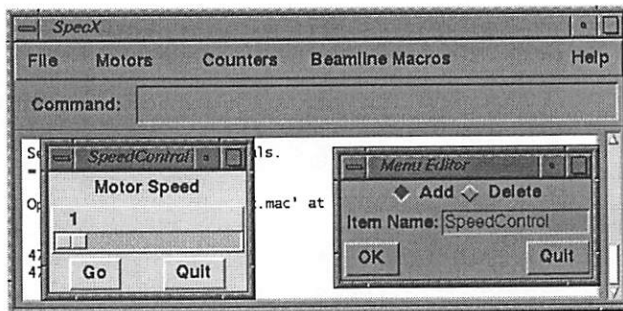


Figure 5. Extending the program with EditMenu

widgets. To demonstrate how to implement communications between the added Tcl/Tk screens and the underlying program, the "SpeedControl" screen (Figure 5) will have its scale widget variable value retrieved and used to control the update rate of the second screen. Listed below is the program macro that communicates with these two interfaces:

```
def RunMotor '
for (i=0; i<50; i++) {
#the following 6 lines are used to
#drive an imaginary device, which
#in this case is another Tcl/Tk
#screen
    var1 = (i*20)%100
    var2 = 100 - (i*20)%100
    var3 = (i*10)%100
    sendgui "RunningMotor" "var1"
    var1
    sendgui "RunningMotor" "var2"
    var2
    sendgui "RunningMotor" "var3"
    var3
    guival "SpeedControl" "speed"
    sleep_time = input()
```

```
sleep_time = 1 - sleep_time/10
sleep(sleep_time);
}
```

In this example, we did not edit any of the code generated by SpecTcl. Our only requirement is that we rename the generated Tcl file to be the project name used by SpecTcl.

Below is a macro code segment that utilizes "popgui" and "gui_exec":

```
def get_motor_code '
{
    local i
    popgui "MotorSelect"
    for (i=0; i<MOTORS; i++) {
        code =
        sprintf(".listframe.motors
insert end %s", motor_mne(i))
        gui_exec "MotorSelect" code
    }
}
```

The "Motor Select" screen that it communicates with is seen in Figure 6. This is a realistic example. A macro performs some operation on some motor. Instead of prompting the users with a text string and forcing them to respond at the command line it would be better to present the user with a list of all motors, and allow them to use their mouse to select the desired one. Since the list is really used more like a dialog as opposed to an interface for execution of a function, we do not need to see it under the Beamline Macros menu. Instead we will call "popgui" from the macro to display the listbox when needed.

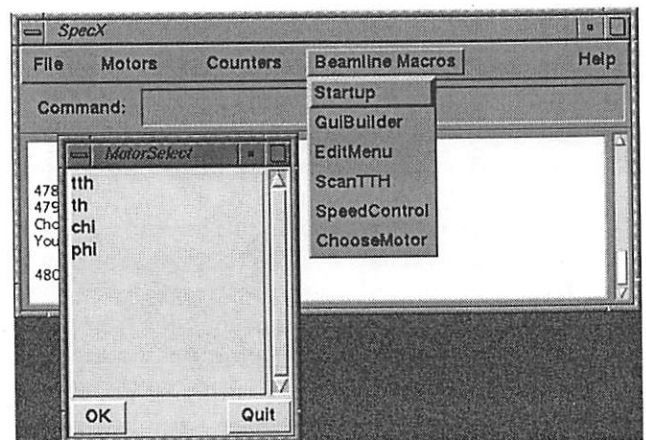


Figure 6. MotorSelect listbox

By selecting "delete" in the EditMenu screen, one may remove any of the program extensions that were added with GuiBuilder or EditMenu.

Implementation Details

Figure 7 illustrates the architecture of the system. The basic GUI, which is a C program, forks off the instrument-control program, and communicates through the standard I/O channels that have been redirected to pipes connecting the processes. An extra pipe is added that is used for the passing of information from the underlying program to the GUI. This information is parsed by the GUI and used to update interface components. We have used this communications scheme to convert other tty-based programs to be GUI-driven and have described the details in [4].

"GuiBuilder" and "EditMenu" are executed with UNIX "system" calls. The Tcl/Tk screen extensions are indirectly executed with the Unix "popen" call, and the returned file descriptors are saved in an array of records that contain screen names and their corresponding file descriptors. Since the basic GUI's standard output has been redirected to the underlying program, it follows that any program spawned from the GUI will automatically have its standard output directed to the child program as well. Therefore, communications from the Tcl/Tk screens to the underlying program are accomplished by simply writing and flushing stdout.

The flow of information from the instrument-control program, through the main GUI, to the Tcl/Tk screen is a bit more complicated. As mentioned above, the Tcl/Tk scripts are popen'd "indirectly". We have written a Tcl/Tk program that we have named "tcl_bridge" that allows us to access and control other Tcl/Tk programs. tcl_bridge performs a *fileevent* call on its standard input and registers a procedure to parse the standard input and do what we request based on commands that we have defined. The last line of tcl_bridge calls Tcl's "source" command with the name of the script that we originally wanted to execute. For example, the result of a call such as "tcl_bridge mytclprog", is whatever "mytclprog" did before plus its monitoring of its standard input for requests to update variables, print variable values, or execute new code segments that we send it. More will be said about tcl_bridge later.

The two-way communications between the child program and the Tcl/Tk screens are accomplished as

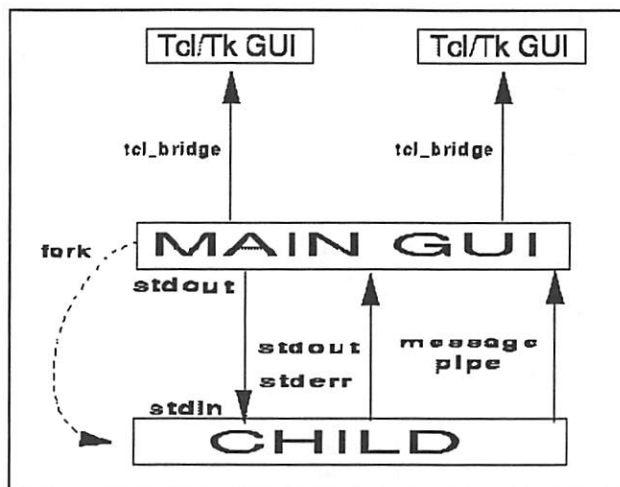


Figure 7. System architecture

follows: To retrieve a value from a Tcl/Tk screen variable, the macro makes a call to "guival" with the name of the screen and the variable value from that screen that we wish to retrieve. "guival" is a macro that will put this information into a format that is recognized by the main GUI as a request to be forwarded to a Tcl/Tk script. The main GUI will traverse the list of active Tcl/Tk scripts and forward the message to the appropriate one. The script will then parse the message and take the desired action, in this case, writing out the value of the variable that was requested. The underlying program can then retrieve this value with a read statement immediately following the guival call. The sendgui call behaves in a similar manner but results in a different request being sent to the Tcl/Tk script. In this case the variable value will be updated instead of output. The "gui_exec" call behaves as the previous two but results in its second argument being executed with an "eval" call in the target Tcl/Tk script.

Saving Extensions

Currently, all GUI extension screens are saved by saving their corresponding Tcl/Tk scripts in the directory in which the main GUI is running. In addition, a file listing these extensions is created, maintained, and read by the main GUI so that the extensions will be present upon subsequent executions. It would be easy to change this scheme but it has worked well for us so far due to the fact that our users typically run these instrument-control programs from the same directory each time. Another approach might be to have an environment variable point to a list of directories in which extensions and their databases may be stored.

Tclbridge vs. Send

When we considered how we might implement the communications between the Tcl/Tk applets and the rest of the system we initially thought that Tk's "send" command would be the answer. However, using the send command required us to switch all workstations using our software to xauth-style server authorization, or to compile Tk with its security flags off so that it would allow the use of "send" in an insecure manner. We consulted with a number of colleagues on the possibility of converting from xhost to xauth-style server authorization. We found that most people had heard of xauth, few knew how to set it up, and no one was eager to change from what they had been using. Based on this we felt that requiring xauth's use would be a major obstacle in the program's use at research stations that we did not administer. Furthermore, the idea of having Tk compiled with its security flag off on government computers controlling, in some cases, millions of dollars worth of instrumentation was clearly not acceptable. Therefore we decided to develop our own method of controlling and accessing data from Tcl/Tk programs. The resulting tcl_bridge has provided us with a reliable and secure solution without dictating how we permit access to our servers.

A Generic Version

In the application that was just described, we developed macros for the instrument control programs that allow for the easy access and control of Tcl/Tk screens. We have also developed a package of C routines that accomplish the same goals. These routines are used in concert with a "boiled-down" generic version of our GUI that, for example, can be used to rapidly develop an extensible GUI for an existing command-line driven program. Typing "xrun <some_program>" will result in the program specified being controlled by the XRun GUI. Commands typed into the command window will be communicated to the underlying program with the output from that program being displayed in the scrolled window of XRun. In addition, the same facilities for communication with Tcl/Tk in the instrument-control program are available in this version. For example, one may call the "sendgui" C routine from the child program to update screens generated by GuiBuilder or incorporated with EditMenu. Figure 8 displays XRun controlling an unmodified version of gnuplot [5]. As a simple illustration, a trivial screen calling gnuplot's "plot" command was generated with GuiBuilder and can be used to modify the graph displayed by gnuplot. Using XRun, converting a command-line driven C program to

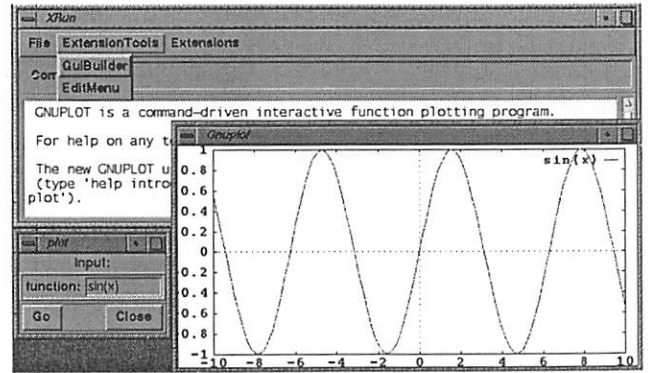


Figure 8. XRun controlling Gnuplot

one that is controlled by a user-extensible GUI should only require minor modifications to the original C program source. This is the same approach that we have used in developing the instrument-control GUI described previously as well as other programs in use at our facility. We have also used a modified version of the C code that provides the communications with the Tcl/Tk screens via tcl_bridge, to develop a Tcl/Tk front end for a robot control program developed at Carnegie Mellon University.

Tcl_bridge and Perl

We have also found tcl_bridge to be very useful as a means of interfacing Perl with Tcl/Tk. We liked Perl for executing and interpreting the output of our data-analysis programs and generating reports and command files based on the output. At some points of the execution of these Perl scripts, we wanted to display information and solicit user input in a graphical fashion. It was trivial for us to write Perl routines that could use tcl_bridge to run, access, and control Tcl/Tk scripts. We looked briefly at the possibility of using TkPerl or PerlTk for these tasks, but found that we could accomplish everything that we wanted by using tcl_bridge to provide the interface between Perl and Tcl/Tk. We liked this solution because it was not dependent on any particular version of Perl or Tcl/Tk. Also, the Perl and Tcl/Tk scripts were separate and developed independently of each other, and we did not need to learn a new syntax which looked like a hybrid of the two languages as appeared to be the case with TkPerl and PerlTk.

Conclusions and Experiences

This work was initiated to solve the specific problem of providing a user-friendly front end that could be easily extended by others, irrespective of their

programming abilities, and without the need for its author to intervene. We are confident that this effort was successful owing to the fact that we have witnessed users of the program interacting with screens that they have created themselves. Although the codes were at first developed specifically for our instrument control application, we have seen that, particularly in the case of the `tcl_bridge` and associated libraries, their usefulness is quite general. Part of the reason for this is that in many environments few software systems are monolithic. When developing a software extension it is rarely accomplished by modifying the source of one large program. The solution is often to integrate a new component into the system, thus preserving the reliability of the original software. The first choice for a language to implement a new component is often a scripting language such as `Tcl/Tk`, `Expectk`, or `Perl`. This component approach requires an easy way to interface the different processes written in various languages and our method has worked very well for us in this respect in a number of varied applications.

Availability

Documentation and codes described in this paper can be found on the World Wide Web at <http://lsx12e.nsls.bnl.gov/x12c/tcltools.html>. Interested parties may freely copy, use, and modify the code. We ask that those who make use of any of this work please provide us with feedback.

Acknowledgments

We acknowledge the important help and suggestions that have resulted from our interactions with Kate Feng-Berman and Gerry Swislow. They are the respective authors of the instrument control programs `ACE` [6,7] and `SPEC` [8] that the GUI described in this paper was written for. This work was supported by the U.S. Department of Energy under contract DE-AC02-76CH00016 with Associated Universities, Inc. and by a biological instrumentation grant from the National Science Foundation.

References

1. S. Kate Feng, D. Peter Siddons, Lonny Berman, "NSLS Beam Line Data Acquisition and Analysis Computer System". *Nuclear Instruments and Methods in Physics Research A* 347(1994)603-606.
2. Integrated Computer Solutions Inc., "The Builder Xcessory Reference", Cambridge, Ma., 1991

3. Uhler, Stephen, A Graphical User Interface Builder for `Tk`, *Tcl95 Workshop Proceedings*, Toronto, Canada 1995
4. J.M. Skinner, R.S. LaBarca, R.M. Sweet, A User-Extensible Graphical User Interface for control of either instrument-control program, `ACE` or `spec`. *Nuclear Instruments and Methods in Physics Research*, Submitted April, 1996
5. T. Williams, C. Kelley, *GNUPLOT Interactive Plotting Program*, Copyright (C) 1986 - 1993
6. J.M. Skinner, J.W. Pflugrath, and R.M. Sweet, *SHARE 80 Proceedings*, Session I224, (1993) San Francisco, Ca.
7. S. Kate Feng, D. Peter Siddons, John Skinner, Robert M. Sweet, *NSLS Beamline Control and Data Acquisition Computer-System Upgrade*", *Proceedings of ICALEPCS95 conference* (1995), Chicago, Illinois.
8. `spec` is a trademark of Certified Scientific Software, PO Box 390640, Cambridge, MA 02139-0007, tel:617-576-1610, fax:617-497-4242, <http://www.ceritf.com>, info@ceritf.com

Visual Tcl

Building a Distributed MultiPersonality GUI toolkit for tcl

Mike Hopkirk

Santa Cruz Operation Inc

hops@sco.com

<http://www.sco.com/Products/vtcl>

Abstract

This paper details some of the Requirements, Design, Architecture, History and Lessons learnt in implementing Visual Tcl (Vtcl) - A distributed multiple GUI toolkit extension to Tcl oriented initially and mainly at generating User Interfaces for System Administration use.

Why Another GUI for Tcl?

Possibly the first question asked is why would we would even consider another tcl GUI extension given that Tk exists. These were some of our requirements:

- Need for a 'pure' Motif based interface - For better or worse all our existing GUI applications are Motif Toolkit based - our GUI toolkit has to be based on the Motif Toolkit, not a look alike or work alike or 'compatible' toolkit.
- Need a Character interface, We still must continue to support character consoles and terminals.
- Possibility of other Interfaces later.
The GUI interface du jour changes. We want to support other interfaces in the future without having to reimplement the entire package. The most recent example of this is supporting MSWindows
- We want the above without supporting multiple Code bases.
- Higher level of abstraction - embed more policy, Simplify UI decisions. We don't want to burden developers with implementing their own (possibly variant) styles of dialogs components and we want to provide

an environment where the developer can concentrate on the details of the UI rather than Toolkit interaction trivia or details of the sub components from which the UI is composed. (This and the character requirement precludes use of solutions such as TclMotif and Wafe[4]).

- Simple hooks into the native help system. We have the model for a help system that we think is pretty good and needed to provide hooks into it (its both topic and widget tree based). We want to provide something that may be able to plug into the native help system on other platforms.
- Easily learnt, easy to develop, Easily extendable and customisable. We get some of this from using Tcl.
- When this was started there was no complete GUI system for Tcl or scripting system for X and Motif.

History

1990 Decided we needed a Tool for building Administration GUI's. This came from a

requirement to improve the system administration capabilities of our systems and the decisions that the right people to do that were the sysAdmin engineers and that it wasn't reasonable for them to learn Motif/X and Curses programming to the level required.

The original system was built around the Bourne shell, Forms based with very high level of policy enforcement. The programming model was procedural rather than event driven using a blocking 'allow Form filling here' style.

For a number of reasons this turned out to be rather a hard sell. It was hard to organize programs in any reasonable way around the user input state and the design didn't provide the control and flexibility the screen designers felt they needed.

1991 Switched to Tcl base.

We also moved to a GUI model that was more component oriented with much more scripting flexibility and changed the main Form controls to use an event driven paradigm.

This was wildly successful, Use spread like a weed both within the SystemAdmin GUI project and for such things as Installation, application configuration, small user productivity apps and one off GUI needs.

Here was also a period of feature growth vigorously driven by user requests. The base started off as a prototype that was well received and accreted features and functionality.

1993 (July - March 94) The previous accreted prototype was becoming difficult to maintain so we rewrote it. The major user visible modification here was to accommodate a more Tk-like widget

naming scheme. Architecturally we created something of a component or trait inheritance architecture that allowed us to remove significant code duplication and rationalize a lot of the option handling.

We also added a few more things that had been requested that were difficult to accommodate on the previous code base - support configuring widgets after widget creation, More flexibility and control over displaying and hiding forms and dialogs, Allow more explicit control over widget positioning, Make all dialogs non Blocking.

Since there was already an existing working script code base some of our users were not keen on this change. We managed to convince them to take the first steps through a combination of bribery, hand holding, automation aids and threats. Fortunately once they'd started using the rewritten system they became wedded to it...

1994 Mainly a period of consolidation. We put a lot of effort into the Character support library and character interface to increase capability and smooth out scripting differences from the Graphical interface.

May 1995 - Deployment of shipping product using Vtcl. ScoAdmin, Custom and (earlier in late '94) DCE Administration. Result:

~68000 lines ScoAdmin
~41000 lines Custom
(Installation)
~40000 lines DCE admin

End user response seems to have been everything we could hope for - The GUIs themselves have been well received and there has been no negative feedback about it being

implemented using Vtcl (most people haven't noticed or don't care which was the point of the exercise).

Feature Set

Vtcl exposes (in some form) probably about 70% of the Motif Widget set to the scripting interface (exceptions are the Drawing Area Widgets, MainWindow and some of the separate Managers e.g Paned Window). This exposure is generally at a level higher than that of the Motif API and we don't give direct access to any underlying Support APIs like X or Xt (as opposed to interfaces like tclMotif, Wafe [4] and DtKsh) favoring instead more a component only model.

For example we have a formDialog that encapsulates the notions of a TopLevel Shell, a Form (with associated geometry on its children) and capability for providing a set of default buttons. These conform to the style and policies defined by their style Guides (Motif [2]) or where these didn't exist from Human Factors Feedback.

Geometry management is presented in a manner similar to X/Motif with parent widgets giving behavioral constraints on their children. We give a very much simplified interface to the geometry control of the Motif Manager widgets (currently only Form and RowColumn). Most of the apps use Form exclusively.

The form layout simplification is setup such that with no explicit layout directions a 'simple and reasonable' layout is generated, explicit geometry options provide both generic and direct script control, overriding or augmenting the default layout handling.

In addition to the Motif Primitive and Manager Widgets we currently add three new ones, A

ComboBox, A SpinButton (implemented in Vtcl) and a DrawnList. The DrawnList is heavily used, it presents a multicolumn list interface which can contain embedded pictures - this is convenient for displaying pictorial hierarchical trees and visually tagging related items.

We abstract the notion of colors and fonts by a single level of indirection. The script specifies a purpose oriented color or font (e.g. foregroundColor, urgentColor, titleFont) which the server is responsible for mapping onto something that makes sense in that GUI environment.

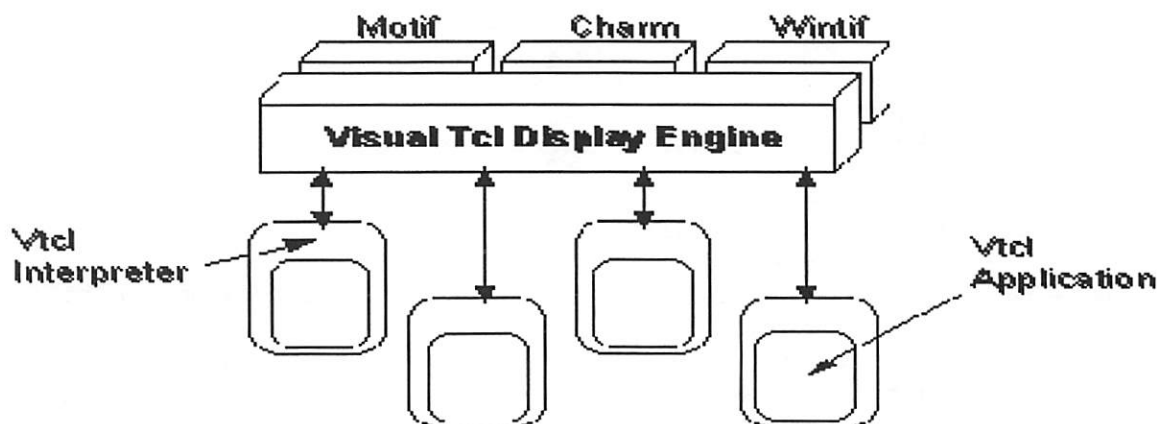
We also extended the Motif notion of displaying bitmaps to support XPM format multicolor pixmaps for those widgets that support pixmaps.

This obviously doesn't work in the character GUI where these are ignored though the script developer may provide an equivalent handling or mapping for that case.

Over and above the widgets we also provide a point help capability and support access to the platform Help system (optimized toward ScoHelp) both explicitly as a script command or implicitly from help Keys, a Help Menu or from Help buttons. This is decoupled from the scripting system.

On the interpreter side the extension supports an event driven state, callbacks firing as a result of a timeout or when nothing else is happening or when an I/O source or sink comes ready.

Architecture



Architecture

Vtcl is a client-server application. We have the notion of a GUI engine or Widget Server engine running autonomously serving out requested user interface applet interfaces as requested by a user. Multiple clients (usually Vtcl interpreters) can connect to and synchronously communicate with the server in order to get a GUI created and rendered.

User Interaction is fed back to the (interpreter) clients at 'interesting' points as messages that eventually fire script callbacks. Additional data is passed to these callbacks providing useful information associated with the callback.

In this model variant GUI's are easily supported by changing the GUI engine that the clients connect to.

The Server keeps its own notion of the separate applet clients on a per connection basis. It does some initial applet hierarchy setup on connection, cleanup on client closedown or on losing the client connection and otherwise responds to client commands by creating widgets or

modifying widget or application hierarchy state.

The server can be thought of as just a rather dumb high level display device, It does not currently contain an embedded interpreter. This is largely a legacy decision and while we may possibly simplify the server implementation we believe we would lose (or face providing server interpreter hooks which would cause loss of) a lot of the current scripting simplicity by doing so.

Client server communication is done using a simple non textual protocol that is basically just a breakdown of the command and options (to the server) and a text string or array of text strings in reply (Callback, errors and command acknowledgment or result).

As an implementation detail this was originally over named pipes, We now have the capability of supporting both that and remote socket connections.

Scripting Interface

Vtcl is obviously based on a core tcl interpreter [1] - this has been extended with two extra packages:

TclX [3] which gives us access to POSIX system interfaces, Debugging and development commands, Message Catalogs, Keyed lists and extra File String and List commands

and the

Vtcl extension itself which are just the commands that provide the graphical interface.

On top of this we provide a package library of convenience routines that encapsulate some common functionality, provide a yet higher level interface to some of the widgets and do commonly required things.

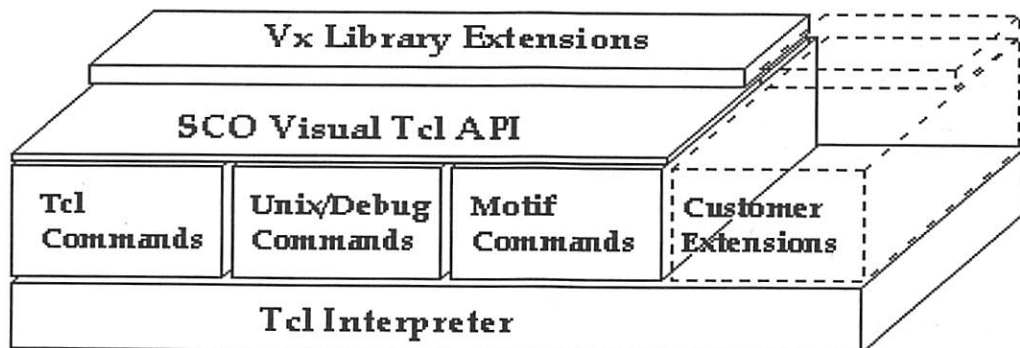
In many cases the applications using Vtcl also provide their own package libraries that encapsulate Vtcl components to give their own application specific functionality.

For the System administration functions the interpreter itself is augmented with additional commands to provide access to our System administration Framework.

With regard to Vtcl alone, GUI manipulation is connection oriented: GUI creation, manipulation and interaction happens within the bounds of a VtOpen and VtClose command (or the exit of the interpreter). There are no concurrent connections from a single interpreter though multiple sequential connections are possible

There is a single command for creation of each type of Component and two additional commands for generic access and setting of component state. Some components have additional commands for accessing or changing that specific components state.

An Extensible Interpreter



Scripting Example

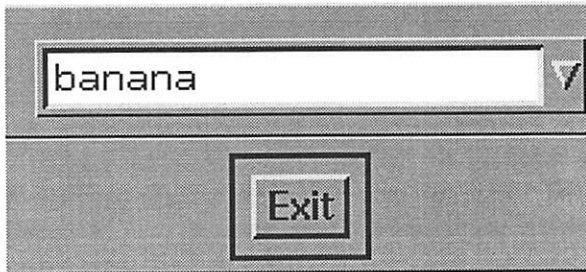
Heres a script example to give some idea of the 'flavor'

```
#!/u1/hops/bin/vtcl
set app [ VtOpen Example ]           ;# Open Server connection

# Make a dialog with single default controlling button
set f [ VtFormDialog $app.f -title "Tcl/Tk WkShop eg" \
        -okCallback myQuitCB -okLabel Exit \
    ]
# declare the callback proc for the 'interesting' action done on the combo box
proc myQuitCB cbs {
    global combo
    echo [VtGetValues $combo -value]    ;# same info avail in cbs structure
    VtClose                             ;# Close connection - shutdown application
    exit                                ;# exit script
}

# Create a ComboBox in the dialog - list contains 3 items
set combo [ VtComboBox $f.combo -itemList { "banana" "plum" "apple" } ]
VtShow $f      ;# Display the Dialog for the first time
VtMainLoop     ;# Accept events
```

Looks like this when displayed using the X/Motif Display engine



Heres the same script displayed using the character engine



Interaction and callbacks

Interaction is 'event' driven from the point a script enters the Mainloop, prior to that the script usually describes the widget hierarchy and state, sets up callback actions and makes the hierarchy visible.

Tcl Commands (usually procs) are registered on the widgets for feedback to the application of interesting actions - These are not presented directly as user events though in most cases that's where they come from.

All widgets that allow callbacks support a "-callback" option which is the callback for the 'normal' or 'usual' interesting action of that widget. e.g. button pressed, list element selected. There are other callback options depending on the widgetClass - these either allow access to a variant action or to somewhat lower level of interaction.

There are also a number of additional callbacks that are not directly widget or user interaction related which allow added control over the hooks into the help system and trapping of errors in callback commands.

All callbacks are passed a TclX keyed list containing information pertaining to the callback: Common items are usually

dialog - dialogName the callback occurred in

widget - the widget the callback occurred on

value - The 'value' of the widget when the callback fired

Additional or variant items may also exist depending on the callback

mode - a hint of the presumed user intention that fired the callback e.g. done, changed, select, selectSame.

itemPosition - positions of selected items (list)

Widget Naming

Widget naming is hierarchical with each level dot separated. A widget name includes its entire parent hierarchy

e.g. .app.form.btn

and if its required to be referenced later should be unique. The server enforces this though it can be told to allow a name as a duplicate of an existing widget. Each Widget Command returns the name of the widget created given the new widgets parent and desired name. The server may add its own (internal) widgets into the hierarchy as part of a widget command (this is an irritating Motif-ism we haven't attempted to hide and is most noticeable for Motif Dialogs) which means that the name you give a widget may not always be the name you end up with.

Vtcl Commands

GUI Commands are action-object oriented and generally of the form :

```
Command objectName \  
    [-option [arg] ... ]
```

We've tried to provide a high level interface to components with a lot of functionality that the options trigger. There are no Key or event binding commands for example - we rely on the bindings of the native GUI component and try and present the widgets as a 'black box' with actions at a functional rather than user level.

The commands breakout into 20 Widget Commands (2 Managers), 21 Widget Manipulation commands, 3 Connection Oriented, 7 miscellaneous and 7 Interpreter only.

Widget Options

Widget options are used to trigger or describe widget actions or state (You could think of the Widget as a Class and the options as setting attributes and notification methods on that class though we make no effort to present it in those

terms).

The options are categorized into several groups depending on their argument types. Options are recognized and filtered, and option arguments are validated for syntax by the interpreter, Semantics are recognized (or ignored) by the server. This implies unrecognized options are discovered early without server interaction, misplaced options rely on the server to handle.

The options (and widget capability) are presented as a Class hierarchy from a base OBJECT class supporting common options with metaclasses for DIALOG, FORM, ROWCOLUMN, LIST, LABEL and BUTTON.

Form geometry management capabilities are presented as an inheritance of the GEOMETRY metaclass on the forms children. These specify where the children are positioned in the form relative to the form and their siblings.

Convenience routines ('Vx' routines)

These are tcl procs in a package library and provide the following:

1. Another layer of abstraction over some Vtcl functionality.
E.g. Menu/OptionMenu - provides specification of a menu hierarchy from a simple data structure
2. Added capability or special functionality:
e.g. Aligned labelled widgets,
Centering and aligning
widgets in Forms,
SpinButton Widget
3. Common/Useful functions:
Widgets 'basename', ExitForm/
QuitApplication callbacks,
Association of variables with
widgets. Keyed list display,
Static variables.

Some Other Issues

CHARM - the Character Support Library

As a result of an earlier project we had the beginnings of an X/Xt/Motif compatible library called CHARM (Character Motif) that mapped most of Motif and a subset of X/Xt onto a Curses interface. This library also mapped keyboard input and redraw processing into an Xlib like model.

We managed to avoid a character specific version of the server code by using this library and writing the server to a single source code base as a normal X/Motif application then addressing any differences as they arose.

These mainly entailed augmenting the CHARM library and headers where necessary to provide a more compatible Motif/X API and special casing any places where we couldn't map or hide any differences in the two interfaces.

There is a single code base for both Character and Motif Servers of which about 5% is special case code for either of the two environments. The majority of this is additional code for utilizing graphical capability not applicable to a character environment.

Determining a Unique Server

The graphical server invocation is unique per uid/LANG/DISPLAY triplet. This server persists until explicitly shutdown or the user logs out.

The character server invocation is unique per uid/LANG/tty - its difficult to get coarser granularity than this without building some form of tty redirection capability onto curses. This gives Vtcl programs spawned from each other (on the same tty) use of the same server.

Startup

On VtOpen the interpreter attempts to detect its corresponding Server for the environment its running in and GUI desired (by testing a "well known point" - existence of a named pipe or success in binding to a socket). If a Server is detected the interpreter sends an Open message to the detection point.

The server, once it is up and running, listens on a 'well known point' for connections. On a connection the server saves some internal state (communication file descriptors) and creates a toplevel shell for the application which is returned to the client (interpreter).

If the server is not detected the interpreter forks and execs it (for the local case). There is an extra level of handshaking here while the server opens a Display connection, After it has done so it sends an ack (or fail) message to the client and continues on with the normal applet open sequence.

So far we've punted on the remote execution server startup and assumed its already running. A better solution would be some sort of host local registry and execution server running from inetd or equivalent that records what servers it may have started on that host and passes back their connection information or starts ones that don't currently exist...

Applet shutdown

A client close message or losing the client connection (assumed due to the client exiting) causes destruction of the applet topLevel shell. All of the applet connection state is associated with this shell so cleanup is relatively simple and done with the Shell Destroy Callback handler.

Server MultiThreading

There isn't any. Apparent multiple applet threading is done using event handlers on the file descriptor for the (server) input connection. The event handler parses and services the request and the result is sent back to the client interpreter. There is no applet state shared between applets beyond that embedded in the use of the Motif and X libraries.

Interpreter interface

Apart from seven interpreter local commands (which we can now lose since they're now available in tcl7.5) most of the Vtcl commands messages are implemented as a stub that does nothing more than send a message to the server, wait for a reply (status and optional string) and passes that back to the interpreter.

The Close and DestroyWidget commands require some additional handling to clean up any pending callbacks for those widgets that may not have been serviced yet. FileSelectionBox also has additional handling to set the server side applets notion of the current working directory to the same as that of the interpreter.

The major effort of the command stub is to parse the command (this is driven though a table of commands and a table of options) and marshall command and options data to (or from) a form conformant with the wire protocol.

Callbacks are implemented as an asynchronous message from the server containing a string (the argument to the widgets 'callback' option) and some data detailing the callback information. The callback data is assembled into a TclX keyed list and appended to the callback string. This string is then evaluated in the interpreter at global scope. Any errors in the evaluation result in a search for a registered error

callback and its evaluation with some information about the error or, if none is found, return of an error status to the interpreter which causes an error return from the MainLoop command.

Callbacks received while a command is waiting for a synchronous reply are queued for execution the next time the Mainloop code is executed.

All interpreter side user interaction is serviced from the VtMainLoop command which is essentially a select call inside a loop. It fires any pending queued callbacks, waits for incoming fd activity and invokes the callback, handles signals and connection errors or fires workproc callbacks.

Communications protocol

This is (for historical reasons) a little gnarly - since its integral to the entire system it also has a lot of inertia which impedes changing it.

We use a mixed binary and text protocol organized for the convenience of the Server in breaking out commands, options and arguments. Each protocol message is typed (COMMAND, COMMAND_RETURN, CALLBACK, ERROR) and contains a field count and number of fields count. Each field is also typed and possibly counted (arrays). Field types correspond almost directly to the allowed option types - INTEGER, STRING, STRING_ARRAY, STRING_2D_ARRAY, INTEGER_ARRAY, OPTION (Special sort of string).

This format doesn't give us much beyond some parsing convenience and entails worrying about byte ordering and word sizes. We're debating changing to a fully textual protocol (which would alleviate the above and give some debugging benefits at the possible cost of some decreased security).

Security

The protocol stream is currently relatively easily snooped (for remote connections at least). We're examining the requirement to allow this to be encrypted in some form to protect information being passed across a network.

One latter request has been for encrypting scripts in such a way that they are protected from casual browsing and modification. We have a modified interpreter (and encryption utility) that handles (and creates) such scripts and refuses to run unencrypted files. The encryption algorithm is fairly laughable in terms of uncrackability for anyone who has any idea of what they're doing but the level provided is acceptable for the requirements given.

Eventually I anticipate that the provision of a Tcl compiler will enable us to sidestep this to some extent.

Lessons

What Worked

- Using Tcl
- Client Server - allows selection of GUI engine without affecting interpreter/scripting, distribution of display to a separate node regardless of the capabilities of the Display system and provides a way to minimise the X/Motif application startup time.
- Abstracting Components, Enforcing a Higher level of policy - simplifies scripting and allows concentration on the UI rather than the components to provide it.

Problems

Trading Ease of Access and Flexibility.

There is still constant

tension in this process - some users want more flexibility, more access to capabilities we have not exposed, None want to lose the current level of simplicity or have to involve themselves in more detail. We cheated on this and give a fallout into the underlying toolkit. We've been working on the premise that if a lot of users are using the fall through capability for a particular capability we should promote it into a core supported option - so far it seems to be working.

Timing Issues

Separation of callback execution from User interaction opened up a race condition between User interaction and script modification (disabling) of UI components. This we kludged around with some callback and app locking commands/options. The solution isn't ideal from the script developers point of view since it exposes some recognition of the separation of interaction and execution but the users liked the problem even less and the solution is adequate if not elegant.

Character Engine

We initially treated the character engine as the limiting factor for features/functionality we had to relax this somewhat in order to be able to generate adequate Graphical interfaces (e.g. support of pixmaps). It turned out that ignoring functionality due to graphical constraints (e.g. pictures in a character environment) and allowing script override works out well enough.

The supporting library we use instead of Xt and Motif in this environment was originally implemented as 'Motif on Character'. This model turns out to be inadequate without a mouse and being character the user expectations of capability (especially with respect

to navigation) are different from Motif. The library is migrating more towards a character interface with a Motif API.

Mistakes

In hindsight we believe we made some mistakes in our presentation of the user model and functionality:

The GUI command and option model is very X/Motif-centric in orientation (due mainly the use of interface libraries based around Motif).

e.g. menu construction is an exact copy of Motifs.

In many cases the naming of options is also similar to Motifs somewhat obscure choice of resource names. We would have been better off providing specific widget commands for some of these concepts and being a bit more generic about some of our naming. We also ended heavily 'convenience' command oriented principally with List manipulation. A more consistent use of value setting options or better subcommand orientation would have reduced the initial number of commands to become familiar with.

Due to some late decisions we have somewhat inconsistent levels of abstractions between font and color and pixmap handling, the latter is not really abstracted at all (being file oriented) yet.

There is a large area of totally missing functionality with Window Manager style manipulation and access to selections and the clipboard. Most users also have their opinion of additional widgets we should support (fortunately there are a few points of common agreement).

Finally one thing we totally missed was providing more

compatibility in command model with Tk when we switched to the Tk-like naming scheme. We should have also switched (or at least supported) something more object+option oriented in order to make knowledge transfer between the tcl GUIs much easier.

Deployment and Future Work

As mentioned earlier Vtcl is currently deployed as part of the base technology in SCO Open Server 5. Ports (of the graphical system only unfortunately) are available for SunOS/Solaris, HP/UX, IBM AIX, SGI IRIX, and Digital UNIX. It will also continue to be base Technology in the Merged UnixWare/OpenServer OS.

Apart from addressing some of the deficiencies mentioned above we're also working on some of the following areas:

A GUI builder Tool (in Vtcl using Vtcl).

MS Windows display Engine - Communicates remotely from an interpreter running on a UNIX Server system. The interpreter will eventually follow (sometime after Mark gets TclX ported).

We're also pursuing providing a similar scenario with the Display engine as a downloadable Web applet allowing us to use any suitable enabled WebBrowser as an engine for Vtcl applications.

Since we're always getting requests for added widgets and we don't want to end up supporting every widget ever created we intend to add an extensibility framework to the GUI engine to allow widgets and supporting code to be added dynamically.

Along with the GUI builder we want to provide some additional support for a scripting development environment - mainly better script debugging, profiling and introspection features.

One of the latter requirements

has been to get up to date with the later revs of Tcl and examine what sorts of benefit we can gain from added capability there. The tcl load command has especially interesting possibilities.

Availability

Vtcl is provided on SCO Open Server 5.0 The ports are available as TLS's from sosco.sco.com or as part of Premier Motif.

Theres a Vtcl Web Page at

<http://www.sco.com/Products/vtcl/>

giving somewhat more introductory material than I've covered above, links to the TLS's and ports and links to other presentations and tutorials and general information. It'll probably also have a copy of this paper.

(The above web page is also referenced from the tcl Web page at <http://www.sco.com/Technology/tcl/Tcl.html>)

I can be contacted for comments, opinions or technical info as hops@sco.com.

Source code is freely available on signing a license form by contacting chrisr@sco.com.

References

1. John K Ousterhout, "Tcl and the Tk Toolkit" Addison-Wesley 1994
2. Open Software Foundation, "Motif Style Guide Rev 1.2" 1993
3. Karl Lehenbauer, Mark Diekhans, "Extended Tcl (TclX)"
4. "Tcl Extensions"
<http://www.sco.com/Technology/Tcl.html#Tcl-GUI>

An On-the-fly Bytecode Compiler for Tcl

Brian T. Lewis

brian.lewis@sun.com

*Sun Microsystems Laboratories
2550 Garcia Avenue, M/S MTV29-232
Mountain View, California 94043*

Abstract

To improve the speed of interpreting Tcl programs, we are developing an on-the-fly bytecode compiler as part of the Tcl project at Sun Microsystems Laboratories. This new compilation system supports dual-ported objects that are stored in Tcl variables and passed to procedures instead of strings. These objects allow faster integer, list, and other operations by including an appropriate internal representation in addition to a string. Early performance results show significant improvement for some scripts. On a 167MHz UltraSPARC 1¹, lindex of the last element of a 100 element list takes 3 microseconds compared to 72 for the current Tcl interpreter. A set command that stores a new value in a local variable now takes less than 1 microsecond vs. 5.8 to 10 microseconds (depending on the length of the variable name) for the current system. This paper describes the design of the compiler and its current state, outlines its development plan, and gives some early performance results. It also describes some implications of the compiler for Tcl script and extension writers, and describes how to best take advantage of the compiler.

1 Introduction

Although the current Tcl interpreter is fast enough for most Tcl uses, there are many applications that need greater speed. The traditional approach to improving a Tcl program's performance has been to recode critical portions in C. While this is effective, it is awkward. Also, an increasing number of demanding applications such as the exmh mail user interface [Welch95] are being written entirely in Tcl. Recoding in C also makes the development of portable applications much harder. A significant advantage of Tcl7.5 is that it allows programmers to write scripts that can run unchanged on UNIX®, PC, and Macintosh systems.

1. UltraSPARC and Java are registered trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark, licensed exclusively through X/Open Company Limited.

The two main sources of performance problems in the current Tcl system (Tcl 7.5) are script reparsing and conversions between strings and other data representations. The current interpreter spends as much as 50% of its time parsing. It reparses the body of a loop, for example, on each iteration. Data conversions also consume a great deal of time. Adam Sah [Sah94] found that 92% of the time in incr's command procedure Tcl_IncrCmd() was spent converting between strings and integers. Many Tcl programmers avoid using lists today because they know that operations on lists are slow and become slower with long lists. For example, lindex \$a end requires that the entire list be parsed to discover its last element.

A new Tcl compiler and interpreter are being developed at Sun Microsystems Laboratories to improve the speed of Tcl programs. Our goal for the bytecode compiler is to improve the speed for compute intensive Tcl scripts by a factor of 10.

The compiler translates Tcl scripts at program runtime, or *on-the-fly*, into a sequence of bytecoded instructions that are then interpreted. The compiler eliminates most runtime script parsing. It also makes many decisions at compile time that are made now only at runtime. It can tell, for example, whether a variable name refers to a scalar or an array element, and whether it refers to a local or a global variable. It also compiles away many type conversions. As an example, it can recognize whether the argument string specifying the increment amount in an incr command represents a constant integer.

The bytecode interpreter uses *dual-ported* objects extensively. These objects contain both a string and an internal representation appropriate for some data type. For example, a Tcl list is now represented as an object that holds the list's string representation as well as an array of pointers to the objects for each list element. Dual-ported objects avoid most runtime type conversions. They also improve the speed of many operations since an appropriate representation is already available. The compiler itself uses dual-ported objects to cache the bytecodes resulting from the compilation of each script.

An early version of the compiler and interpreter are running. The compiler emits a subset of the instructions that will eventually be supported. The current instructions directly implement variable substitutions and the `set`, `incr`, and `while` commands. Other commands are supported using instructions that invoke the associated C command procedures. These instructions push and concatenate the dual-ported objects that hold command arguments and results. The largest single item of work remaining on the compiler is to compile Tcl expressions and the `expr` command. Most control structure commands use expressions so compiling them should significantly reduce the execution time of almost every script.¹ Other remaining work includes compiling performance-critical commands like `for` and `lindex` into inline sequences of instructions specific to those commands.

The bytecode compiler and interpreter pass most Tcl regression tests. Tests that fail depend on the specific contents of traceback information in error messages or on the exact formatting of the result of list operations. The bytecode system makes fewer recursive calls to `Tcl_Eval` so error tracebacks now have fewer intermediate levels. The new list implementation consistently uses `Tcl_Merge` to regenerate a list object's string representation, while the traditional Tcl system typically ignores portions of strings not directly modified in a list operation. This can lead to such differences as whether sublists are bracketed with braces or quotes: for example, the result of

```
linsert {a b "c c" d e} 3 1
```

with the new system is

```
a b {c c} 1 d e
```

while the traditional system produces

```
a b "c c" 1 d e
```

The Tcl tests will be updated to reflect the new behavior.

1. Expressions are very expensive today. As an experiment, I implemented a new command `untilzero` that repeats a loop while a variable is nonzero. Executing

```
set x 1000; while {$x>0} {incr x -1}
```

requires 5.1 times more time than

```
set x 1000; untilzero x {incr x -1}
```

(28661 vs. 5577 usec). These two loops do the same work. The difference is the expression that is reparsed on each iteration. (Although `while` commands are compiled into a sequence of instructions, their expressions are still evaluated today by `Tcl_ExprBoolean`).

Table 1 shows performance results for a few simple benchmarks on a 167MHz UltraSPARC 1.

Table 1: Performance Benchmark Results

Benchmark	Time (usec)		Speedup
	Tcl7.5	New	
null proc with 5 args	64	6	10.6
proc incrementing arg	31	7	4.4
proc of just comments	31	3	10.3
set 20 global variables	210	31	6.7
set 20 local variables	206	21	9.8
incr 20 local variables	368	28	13.1
lindex end of long list	134	3	44.6
linsert end of long list	58	12	4.8
iterative factorial	449	265	1.6
list reverse using while	4985	2376	2.0

Some scripts show significant improvement, especially those that make heavy use of procedure arguments and local variables, ones that manipulate lists, and scripts that benefit from not reparsing. Performance improvements for the larger (and more realistic) benchmarks are modest at this time, largely because expressions are not yet compiled. I expect performance to improve significantly as the remaining compiler and interpreter changes are made.

The next section describes the goals for the bytecode compiler. Section 3 describes the compilation strategy. I present the design of the dual-ported object support next. Section 5 and Section 6 give details about the design of the bytecode compiler and interpreter. Memory requirements for the bytecode system are discussed in Section 7. The current state of the compiler is described in Section 8 while Section 9 discusses related work. I explain next why I do not use Java bytecodes. Section 11 discusses implications of the compiler for script and extension writers. The compiler's development plan is outlined in Section 12.

2 Goals for the Compiler

Besides increased execution speed, the compiler's goals include the following:

- Minimize user-visible changes to scripts. Old scripts must run as before with no or very few changes. It isn't possible to change the Tcl language even if those changes would improve the compiler's effectiveness. The compiler must continue to support Tcl features like `traces`, `upvar`, and `unset` even though they slow our implementation considerably.
- Continue to support Tcl's dynamic features like rebinding of core commands and runtime-computed variable and command names. However, we may change the behavior of core commands when those changes allow significant performance improvements without effecting the execution of correctly written scripts: scripts that use the semantics documented in the Tcl man pages. For example, list operations will no longer preserve exact white space between list elements.
- Minimize changes to C code in extensions. I expect to do this by providing new parallel core API procedures that contain the changes. Old C code can continue to use the old interface procedures. Those old interface procedures will often be reimplemented in terms of new functionality.
- Minimize storage requirements during both compilation and execution. Tcl's small memory footprint is a significant feature and must be retained.
- Portability. The compiler and interpreter must run on a wide range of platforms. I cannot generate machine code, for example.

3 The Compilation Strategy

The new compilation system relies upon support for dual-ported objects and a new bytecoded compiler and interpreter. Dual-ported objects are passed to command procedures and are stored in variables. Objects contain a string as well as an internal representation. They reduce conversions by holding an appropriate representation such as an array of element pointers for a list. Although objects contain an internal representation, their semantics are defined in terms of strings: an up-to-date string can always be obtained, and any change to the object will be reflected in that string when the object's string value is fetched. Objects are typed. An object's type reflects the set of operations on its internal representation. The set of types is extensible. Several types are predefined in the Tcl core including integer, double, list, and bytecode.

Compilation in the new system is done as needed, or on-the-fly. When a script is evaluated (say as the result of a call to `Tcl_Eval`), it is compiled into bytecodes that are then executed. We use the term *code unit* to describe the collection of bytecode instructions and related infor-

mation that results from compiling a script. A new Tcl API procedure, `Tcl_EvalObj`, operates much like `Tcl_Eval` to evaluate a script but takes a Tcl object instead of a string; it compiles the object's string value and caches the resulting code unit as its internal representation to avoid later recompilations. The compiler generates instructions for an idealized Tcl virtual machine. This machine is stack-based since this allows programs to be represented more compactly; the encoding of most instructions is a single byte. Since programs are compiled, script parsing at execution time is rarely necessary. Some runtime parsing is needed since Tcl scripts can compute new scripts that they later evaluate. Such runtime-created scripts are also compiled on-the-fly. The compiler will eventually generate bytecodes for most of Tcl's core commands. It can make decisions now made by the traditional Tcl interpreter at runtime. For example, the compiler assigns frame offsets to local variables in procedures to avoid the runtime hashtable lookup done for them in the traditional system.

4 Design of dual-ported objects

4.1 The `Tcl_Obj` structure

Dual-ported objects are used throughout the new Tcl system to hold scripts, strings, integers, arrays, lists, etc. For example, command procedures now take an "objv" array of pointers to the argument objects. An object has two representations: a string and an internal form. Objects are represented by `Tcl_Obj` structures allocated on the heap.

The definition of the `Tcl_Obj` structure is shown in Figure 1. This structure is five words: the reference count, a pointer to the object's type structure, a string pointer, and two words used by the type. The string is the object's string representation which is also allocated on the heap. The two words managed by the type hold the object's internal representation: an integer, a double-precision floating point number, two arbitrary words, or a pointer to a value containing additional information needed by the object's type to represent the object. A list object, for example, contains a pointer to a structure with an array of pointers to the objects for the list elements. An integer object contains an integer value.

At least one of an object's representations is valid (non-NULL) at any time. Representations are computed lazily, when they are needed. An object that contains only a string and is (so far) untyped has a NULL `typePtr`. As an example of the lifetime of an object, consider the following sequence of commands:

```

typedef struct Tcl_Obj {
    int refCount;           /* When 0 the object will be freed. */
    char *string;           /* The Tcl_Obj's string representation. */
    Tcl_ObjType *typePtr;   /* Reflects the object's type. */
    union {                 /* The internal representation. */
        int intValue;       /* -An integer value. */
        double doubleValue; /* -A double-precision floating value. */
        VOID *otherValuePtr; /* -Another, type-specific value. */
        struct {            /* -The value as two words (ints). */
            int field1;
            int field2;
        } twoIntValue;
    } internalRep;
} Tcl_Obj;

```

Figure 1: Definition of the Tcl object structure

```
% set A 123
```

This assigns A to an integer object whose internal representation is the integer 123. Its string representation is left NULL to avoid allocating a string on the heap; if the string is needed later, it can be regenerated from the integer¹. The `typePtr` points to the structure describing the integer type.

```
% puts "A is $A"
```

A's string representation is needed. It is computed from the object's internal representation. Afterwards, A's internal representation holds the integer 123 and its string representation points to "123". Both representations are now valid.

```
% incr A
```

The `incr` command increments the object's integer internal representation and invalidates (sets NULL) its string representation is since it is no longer valid.

```
% puts "A is now $A"
```

The string representation of A's object is needed and is recomputed. The string representation now points to "124".

An object's internal form is typically computed on the first type-specific operation, or when an object is converted to a new type. The string is invalidated (set NULL) when the internal representation is changed, and vice-versa. The string representation is only regenerated when necessary. For example, the string representation of a `for` loop's index variable will never be recomputed unless it is actually used as a string. I expect that almost all

objects will remain a single type, perhaps after an initial conversion.

4.2 Object types

The set of object types is open ended. The Tcl core pre-defines six object types: integer, double, list, bytecode, boolean, and command name. We expect to create many new types in the future. For example, the Tcl core could use a file pathname type to store a canonical platform-independent representation of a file's path. Also, Tk might use objects to store options for Tk commands.

A Tcl object type is defined by a structure containing pointers to four procedures called by the generic Tcl object code. The definition of this type is shown in Figure 2.

The `Tcl_UpdateStringProc` updates an object's string representation from its internal representation. A type's `Tcl_DupInternalRepProc` and its `Tcl_FreeInternalRepProc`, respectively, duplicate and free an object's internal representation. The final procedure, the `Tcl_SetFromAnyProc`, converts an object from another type by producing this type's internal representation. It can always do this by first updating the object's string representation (if necessary) then generating the internal representation from the string. However, the `Tcl_SetFromAnyProcs` for most object types include special case conversions from some number of other types. An example is the double type's `Tcl_SetFromAnyProc`. This supports faster integer to double conversions by directly converting the integer that is an integer object's internal representation to a double-precision floating point number; it does not regenerate the string representation and then parse it.

As an important optimization, an empty string is represented by an object with a NULL string pointer and

1. This optimization is possible only when the correct string representation can be regenerated. It can't be used, for example, for the string "000123" since a later command might depend on the leading zero characters.

```

typedef int (Tcl_SetFromAnyProc) (Tcl_Interp *interp, Tcl_Obj *objPtr);
typedef void (Tcl_UpdateStringProc) (Tcl_Interp *interp, Tcl_Obj *objPtr);
typedef void (Tcl_DupInternalRepProc) (Tcl_Obj *srcPtr, Tcl_Obj *dupPtr);
typedef void (Tcl_FreeInternalRepProc) (Tcl_Obj *objPtr);

typedef struct Tcl_ObjType {
    char *name; /* Name of the object type, e.g. "int" or "list". */
    Tcl_FreeInternalRepProc *freeIntRepProc;
                /* Frees any storage for the type's internal representation. */
    Tcl_DupInternalRepProc *dupIntRepProc;
                /* Creates a new object as a copy of an existing object. */
    Tcl_UpdateStringProc *updateStringProc;
                /* Updates the string rep. from the type's internal rep. */
    Tcl_SetFromAnyProc *setFromAnyProc;
                /* Converts the object's old internal rep. to this type. */
} Tcl_ObjType;

```

Figure 2: The definition of a Tcl object type

typePtr. Empty strings are common and this optimization helps to reduce storage requirements.

The list type maintains for each list object an array of pointers to the Tcl objects that represent the list's elements. This internal representation allows for fast indexing and append operations (which we believe to be the most common) at the expense of slightly slower insertions into the middle of a list. For example, `lindex` is now a constant time operation; extracting the last element of a list now requires only 3 usec regardless of the list's length while Tcl7.5 takes 15 usec for a 10 element list, 37 usec for a 40 element list, and 79 usec for a 100 element list. `linsert` is also faster; inserting an element at the end of a 60 element list is 4.8 times faster (12 vs. 58 usec).

The element array of a list is initially allocated just large enough to hold the list's elements. However, if a list is grown by, say, an append operation, a new array is allocated that is larger than is actually required by the operation. This overallocation improves the speed of subsequent append or insertion operations. When the list type's `Tcl_SetFromAnyProc` generates the internal representation for a list, it parses the entire list. This means that operations on some lists will fail in the new system that would have succeeded in Tcl7.5: if a list has a syntax error after the elements being operated on, the new system will return an error message where Tcl7.5 would have ignored the bad syntax.

The command name object type is used by the bytecode interpreter to cache the result of command hashtable lookups. Hashtable lookups are expensive (about 1 usec on a UltraSPARC 1, or the same time needed to set a local variable in the bytecode system), so avoiding them on

most command invocations significantly improves execution time.

4.3 Storage management of objects

Tcl objects are allocated on the heap. A custom allocator reduces the cost of allocating and freeing objects by maintaining a private list of available free objects.

Because many objects are simply passed as arguments to called procedures, objects are shared as much as possible. This significantly reduces storage requirements because some objects such as long lists are very large. Also, most Tcl values are only read and never modified. This is especially true for procedure arguments, and argument objects can be shared between the caller and the called procedure. Assignment and argument binding is done by simply assigning a pointer to the value. It isn't necessary to copy (and allocate storage for) the entire value. But this raises the problem of knowing when it is safe to free an object. I use reference counting to determine when it is safe to deallocate an object; an object can be freed when the number of references to it drops to zero. I can't use a garbage collector because it would increase Tcl code and runtime memory usage too much.

One advantage of reference counts is that they support an important optimization called *copy-on-write*. Since objects are shared, a new copy must be made before modifying an object. But if an object is unshared—that is, if it has a reference count of one—the object can be modified directly without having to make a copy. Copy on write reduces storage requirements and execution time.

<pre> push1 <1 byte index> push4 <4 byte index> pop concat <1 byte count> invokeStk1 <1 byte argument count> invokeStk4 <4 byte argument count> loadScalar1 <1 byte index> loadScalar4 <4 byte index> loadScalarStk storeScalar1 <1 byte index> storeScalar4 <4 byte index> storeScalarStk loadArray1 <1 byte index> loadArray4 <4 byte index> loadArrayStk storeArray1 <1 byte index> storeArray4 <4 byte index> storeArrayStk </pre>	<pre> loadStk storeStk incrScalar1 <1 byte index> incrScalarStk incrArray1 <1 byte index> incrArrayStk incrStk incrScalar1Imm <1 byte index> <incr byte> incrScalarStkImm <signed incr byte> incrArray1Imm <1 byte index> <incr byte> incrArrayStkImm <signed incr byte> incrStkImm <signed incr byte> evalStk jump1 <1 byte signed distance> jump4 <4 byte signed distance> jumpFalse1 <1 byte signed distance> jumpFalse4 <4 byte signed distance> done </pre>
--	--

Figure 3: The current bytecode instructions

5 Design of the bytecode compiler

The compiler is single pass to minimize compilation time. It uses a recursive descent parser that emits instructions for each command as it is parsed.

To hold information needed during compilation, the compiler uses a compilation environment (`CompileEnv`) structure. This holds a code unit's instructions, *object table*, and command location map. The object table is an array of pointers to Tcl objects referenced by instructions. The table has an object for every unique constant in the script that is not "compiled away": for example, the string "A is " needed for the command `puts "A is $A"` above is represented by an object table entry. The command location map has source and bytecode location information for each command. This information is used, for example, to find the source command for a bytecode location. The `CompileEnv` structure also contains a pointer to the current procedure's `Proc` structure (if any) to compile references to local variables, and contains fields that describe the length and other properties of the last command word processed. The `CompileEnv` structure is allocated on the C stack and is large enough to hold the instructions and other information for almost all Tcl scripts. This use of stack-allocated space minimizes the number of costly heap allocations. When compilation is finished, a single heap object is allocated to hold the subset of information required to execute the script.

In order to generate instructions for a command, the compiler first checks whether a compile procedure (`CompileProc`) has been registered for it. This is done just after the command's first word is parsed. If a `Com-`

`pileProc` is found, it is called to generate code for the command. If no `CompileProc` is found, or if the first word involves substitutions that can only be computed at runtime, the compiler emits code to invoke the command's command procedure at execution time. `CompileProcs` exist today for the `set`, `while`, and `incr` commands. Eventually `CompileProcs` will be registered for most core Tcl commands.

At this time, the compiler emits the 35 instructions listed in Figure 3. Some of these implement variable substitutions and the Tcl commands `set`, `incr`, and `while`. The remainder do the work of the traditional Tcl parser by pushing and popping objects, concatenating strings, and calling command procedures. New instructions will be added as more commands are compiled. I expect also that the instruction set will change as I get more experience with the bytecode system.

Most instructions operate on an *evaluation stack*. This stack is separate from the "stack" of Tcl procedure call frames and is also separate from the C call stack. The evaluation stack holds pointers to Tcl objects holding command arguments and results. Each Tcl interpreter has its own evaluation stack. The compiler computes the maximum stack depth needed for each code unit and the interpreter, when starting to execute a code unit, ensures that it has enough stack space. This avoids checking on each instruction whether the stack needs to be grown.

Instructions consist of an opcode byte followed by zero or more operands. Operands are one or four byte integers or indexes. As an example, `push1 <index>` pushes an object onto the evaluation stack. The one byte index refers to one of the first 256 objects in the code unit's object table. Several instructions have four byte variants to

support large scripts, while the one byte variants keep the code for small scripts small. Instructions whose names include the "Stk" suffix take an operand from the evaluation stack.

To make local variables faster, the compiler assigns each local variable an entry in an array of variables stored in a procedure's call frame. This avoids an hashtable lookup on each reference. The compiler also determines whether the variable name refers to a scalar or an array element. These two changes alone make local variable access faster by a factor of 9.5! (From 201 usec to 21 to set 20 locals. Other changes account for a 5 usec improvement.)

Some variables are only created (computed) at runtime. For example, the command `set [gensym] 123` assigns a value to the variable whose name is returned by the procedure `gensym`. To support these runtime computed variables, the compiler emits the instructions `loadStk` and `storeStk` that take the variable name from the top of the evaluation stack.

5.1 Examples of compiled code

Compiling the procedure

```
proc while_1000x {} {
    set x 0
    while {$x<1000} {
        incr x
    }
}
```

generates a code unit with the instructions

```
# set x 0
0  pushl 0          # push object "0"
2  storeScalar1 0   # store into local x
4  pop              # discard value
# while {$x<1000} {\n incr x\n }
5  pushl 1          # push "$x<1000"
7  jumpFalse1 8     # false => goto pc 15
# incr x
9  incrScalar1Imm 0,1# increment local x
12 pop              # discard value
13 jump1 -8         # goto pc 5
15 pushl 2          # while result is ""
17 done
```

The number at the left of each instruction is its bytecode offset. The `pushl 1` instruction at offset 5 pushes a string object containing "`$x<1000`"; the instruction's operand specifies the second object in the code unit's object table. This string is passed to the Tcl expression code at runtime since expressions are not yet compiled. The `storeScalar1 0` at offset 2 stores the object at the top of the evaluation stack into the scalar local variable at offset 0 in the call frame's array of local variables.

This procedure currently runs 1.4 times faster with the bytecoded system than in Tcl 7.5 (26954 vs. 38550 usec). I expect this performance to improve when expressions are compiled.

As a more complex example, the procedure

```
proc lreverse_with_while {a} {
    set b ""
    set i [expr [llength $a] -1]
    while {$i >= 0} {
        lappend b [lindex $a $i]
        incr i -1
    }
    return $b
}
```

generates the instructions

```
# set b ""
0  pushl 0          # push ""
2  storeScalar1 1   # store into local b
4  pop
# set i [expr [llength $a] -1]
5  pushl 1          # push "expr"
7  pushl 2          # push "llength"
9  loadScalar1 0     # load local a
11 invokeStk1 2      # call llength, 2 args
13 pushl 3          # push integer obj -1
15 invokeStk1 3      # call expr, 3 args
17 storeScalar1 2    # store into local i
19 pop
# while {$i >= 0} {\n lappend b [lindex ...
20 pushl 4          # push "$i >= 0"
22 jumpFalse1 23     # false=>goto pc 45
# lappend b [lindex $a $i]
24 pushl 5          # push "lappend"
26 pushl 6          # push "b"
28 pushl 7          # push "lindex"
30 loadScalar1 0     # load local a
32 loadScalar1 2     # load local i
34 invokeStk1 3      # call lindex, 3 args
36 invokeStk1 3      # call lappend, 3 args
38 pop
# incr i -1
39 incrScalar1Imm 2,-1
42 pop
43 jump1 -23        # goto pc 20
45 pushl 0          # push ""
47 pop
# return $b
48 pushl 8          # push "return"
50 loadScalar1 1     # load local b
52 invokeStk1 2      # call return, 2 args
54 done
```

Here the `invokeStk1` instructions are used to invoke command procedures at runtime. In the next few months, the compiler will be modified to emit command-specific instructions inline for most Tcl core commands. This

procedure runs 2.0 times faster with the current bytecoded system than in Tcl 7.5 (2376 vs. 4985 usec for a 60 element list).

5.2 Some compilation problems

a) Variables must be accessed in the correct order

Compiled code must read, write, and delete variables in the correct order. This is because traces must run the correct number of times and in the correct order. Consider the following example:

```
expr{$a} + $b || {$c} + $d
```

The variables must be read in the order b, d, a, then c.

In the traditional Tcl system, the interpreter reads variables b and d when substituting their values. When `expr` is called, it does a second round of substitutions on its arguments itself, and so reads the variables a and c. The order in which variables are read is shown above. Compiled code must read the variables in the same order. I may alter the variable read, write, and delete behavior of some operations to improve the implementation, but I will only do this if the changes do not modify the semantics of those operations or of the `trace` command as described in the Tcl man pages. For example, the traditional Tcl interpreter implements `lappend` using `Tcl_SetVar2` to append each new list element. This triggers read and write traces for each appended element. I may compile code to append the new items all at once and run the traces a single time.

b) `expr`'s substitutions can change the apparent expression

As described above, `expr` does a second round of substitutions on its arguments. This can make the expression's apparent interpretation and the obvious code wrong. Consider the following:

```
% set x 2
% set y {$x+5}
% expr $y*15
=> 77 —this is the correct result but it is
not divisible by 15!
```

From the expression `$y*15` it looks like the final result is a multiple of 15, but this is wrong. `expr` is passed `$x+5*15`, which after `expr`'s second round of substitutions becomes `2+5*15` or 77.

This problem only happens when `expr` does a second round of substitutions. If `expr`'s argument is not enclosed in braces, the best I can do is to generate "optimistic" code for the apparent expression and check at runtime whether this code might be wrong. It can only be wrong if variables substituted in the first round require

more substitutions in the second round. Typically this isn't the case and the interpreter can execute the compiled code. Otherwise, the interpreter needs to back off and invoke `expr` to interpret the expression.

If `expr`'s argument is enclosed in braces, the apparent code is always correct and the test can be dropped. So, expressions protected by braces will execute *faster*. This includes expressions used in `if`, `while`, and other control structure commands.

c) Global variables may not be truly global

In the same way that it currently does for local variables, the compiler could assign each global variable an index in the table of globals and use this index in instructions. It can only do this for variables, however, which it knows to be truly global. Tcl lets a `global` command appear anywhere, including after the use of a local variable with the same name. This is an error, and must be reported as such, so the compiler can only "compile away" global variables known to be global. It can safely do this for `global` commands that appear at the top of a procedure, which is the usual location anyway. Those that appear elsewhere will have to be implemented by a `global` instruction that will do the appropriate checking. This means that `global` commands placed at the top of procedures will be faster.

6 Design of the bytecode interpreter

The bytecode interpreter uses a traditional while loop that switches on the opcode of each instruction:

```
for (;;) {
    opCode = *pc;
    switch (opCode) {
        case INST_INVOKE1:
            ...
    }
}
```

I checked first whether an alternative implementation would be faster. This used an array of procedure pointers, indexed by opcode, to implement each instruction. However, this proved about 20% slower, independent of machine or compiler.

The compiler emits a `done` instruction to terminate the main interpreter loop if no `return` or `error` command is executed. This instruction trades space for time and avoids the need to continually test for the last instruction.

The evaluation stack holds arguments for commands. When invoking a command procedure, the procedure's `objv` array (the array of pointers to argument objects) is set to the address of the evaluation stack element holding a pointer to the object with the command name; no

pointer copying is needed. The interpreter caches a pointer to the top of the stack in a local variable.

If a Tcl program redefines a core command, any code that uses that command must be invalidated. To implement this, the interpreter increments a counter, the *compilation epoch*, whenever a core command is redefined. When a script is compiled, the current compilation epoch is stored in its code unit. Before executing a code unit, the bytecode interpreter checks whether the code unit's epoch matches the current epoch. If not, the interpreter discards the code unit and recompiles its script.

I have reimplemented the command procedures for most commands to be object-based: that is, to take an `objv` array and to return an object result. These object-based command procedures are called directly by the bytecode interpreter. The remaining string-based command procedures are implemented using a wrapper procedure. This wrapper generates an `argv` string array from the string representations for the argument objects, calls the string command procedure, and constructs a string object holding the result. I expect eventually to make all command procedures object-based.

7 Memory requirements for the bytecode system

Strings are a compact way to represent Tcl scripts: no separate instructions or other data representations are needed. The bytecode system improves the speed of executing Tcl scripts at the cost of additional storage for code units and dual-ported objects. How much additional memory is needed?

The body for the procedure `while_1000x` in Section 5.1 is 56 characters. Its code unit requires 18 instruction bytes. Its object table contains pointers to three Tcl objects: an integer object for the source string `"0"` (for which no string is allocated on the heap), an object pointing to `"$X<1000"`, and an empty object representing the result of the `while` command. Since each Tcl object requires five words, the object table requires 80 bytes including the storage for the one heap string. The command location table for this procedure's three commands requires 3 entries of 4 words each, or 48 bytes. So, the total memory for this procedure's code unit is $(18 + 80 + 48)$ or 146 bytes¹, 2.6 times the storage for just the source characters.

The body for the second procedure in Section 5.1, `lreverse_with_while`, is 131 characters. Its code

1. This ignores any overhead words required by the heap implementation.

unit requires 55 instruction bytes. Its object table has nine objects (for `"", "expr", "llength", "-1", "$i >= 0", "lappend", "b", "lindex",` and `"return"`) and requires a total of 261 bytes. There are six commands so the command location table requires 96 bytes. The total memory for this procedure is then $(55 + 261 + 96) = 412$ bytes, or 3.1 times the source size.

Note that five of the nine objects for this code unit were allocated just to hold the names of commands to be invoked by `invokeStk1` commands. One of the main benefits of compiling commands into inline sequences of command-specific instructions may be to reduce the storage needed for programs. In this case, removing just those command name objects would save 155 bytes! The count of instruction bytes would increase a little, but the code unit's storage would still drop to approximately 1.9 times that of the source.

These memory results are preliminary. The actual storage needed for Tcl scripts will change as more commands are compiled inline. I expect to look for further opportunities to reduce memory requirements. For example, it should be possible to find a more compact representation for the command location tables.

8 Compiler status

At this time (May 1996), the basic support and infrastructure for the new bytecode system is complete. The dual-ported object support is finished. The Tcl core implements six object types. Objects are passed to and returned by command procedures and are stored in variables. The compiler emits inline instructions for several key instructions. Support routines exist that allow the development of new `CompileProcs` for commands to be added at the rate of about one a day. The largest remaining item of work is to compile Tcl expressions. Another large work item is to support Tcl namespaces. The specific functionality for namespaces has not been decided, but it will probably be similar to George Howlett's proposal [Howlett94] and Michael McLennan's [incr Tcl] namespace support [McLennan95].

Performance improvements to date are modest for most code: about a factor of two for scripts that use expressions or control structures (since they use expressions). The fact that performance is significantly faster for scripts that make heavy use of variables or lists is promising. The key reasons performance isn't better for all scripts yet include:

- Few commands have command-specific instructions generated for them. A procedure call to a command procedure is still being made for most commands. Also, many objects are pushed, popped, have their

reference counts incremented and decremented just to fabricate the arguments for the command procedures. Appropriate instructions for each command will reduce this significantly.

- `expr` isn't compiled yet. The `expr` command is itself used often and expressions are used in many control structure commands.
- Too many little code units are compiled and executed. This is primarily because control structure commands are not yet directly compiled into a linear sequence of instructions. As an example, an `if` command's `then` and any `else` subcommands are compiled separately, and are executed when the `if` command's command procedure recursively calls `Tcl_EvalObj` on the bytecode objects for their scripts. This results in extra procedure calls and execution overhead as well as extra storage use. This will improve when instructions for those subcommands are emitted inline.
- Repeated compilations. This is because some command procedures are still string-based and can't save the bytecodes of a compiled subcommand in an object. Consider the following:

```
expr {$n*[llength $a]}
```

This is slowed today because the nested command `llength $a` is recompiled, executed, and its bytecode deallocated each time the `expr` is evaluated. This is because `expr` does command substitutions on its arguments, and recursively calls `Tcl_Eval`. This, in turn, compiles the expression but the code unit resulting from the compilation is discarded afterwards. This is a temporary problem that will end when expressions are directly compiled.

- Compilation is expensive at the moment. The cost of compiling `lappend pkgs "stdPkg"` is about twice that of executing it once. Most of the compilation time is spent allocating objects for each word or part of a word in a script. In this script these are the words "`lappend`", "`pkgs`", and "`stdPkg`". When the compiler emits command-specific instructions, most of these allocations will disappear. But even now, the compilation time for most realistic scripts is only a small part of their execution time: a recursive factorial procedure computing the factorial of five spends only 1% of its time compiling.

9 Related work

Adam Sah's TC system [Sah94] provided a speedup of about 5-10 over traditional Tcl. His system introduced the use of dual-ported objects. TC implemented lists using arrays of pointers much as I do. It also used reference counts to decide when to free objects. Like our system

TC used reference counts to implement copy-on-write. This minimized copying in procedure calls and saved a considerable amount of storage. In an attempt to reduce the cost of script execution, TC statically prepared scripts. This did not benefit most Tcl/Tk scripts since most scripts require runtime parsing. Because of this, he suggested instead caching the result of parsing, which is effectively what our system does. His system also implemented several other optimizations, including implementing procedure frames as arrays and compiling variable references into indexes. Unfortunately, Adam Sah never released TC.

Forest Rouse and Wayne Christopher developed the ICE Tcl compiler [Rouse95] that is available from ICEM CFD Engineering. This compiler translates Tcl to C code, which is then compiled. It speeds up typical Tcl/Tk applications by a factor of between 5 and 20. ICE Tcl tracks the dynamic types of Tcl variables in C code using a mechanism similar to our object system. Its `Tcl_Var` structure has fields for integer, double, list, and string representations and includes a flag word that indicates which of these representations is valid. Unlike our system, more than two representations may be valid at any time. This offers the potential for improved speed at the cost of additional memory, greater complexity, and more difficult use. One drawback of translating to C is the significant expansion in application code size (a factor of 20-30 in some cases) and complexity of application development. ICEM has announced plans to develop a bytecode compiler to avoid these problems.

10 Why not use Java bytecodes?

If it proved feasible, using Java bytecodes to implement Tcl would have a number of advantages. The Java virtual machine is widely available (e.g., in the Netscape browser). Using Java bytecodes might also simplify interoperation between Tcl and Java code.

Unfortunately, using the Java virtual machine would be too slow or take too much memory, at least with current Java interpreters. The basic problem is the semantic mismatch between Java bytecodes and Tcl. Consider the Tcl `set` command. Tcl variables behave very differently than Java variables. I can't use a Java instruction like `astore` (store object reference in local variable) to store a Tcl value into a Tcl variable since it doesn't handle by itself such Tcl details as variable traces, `unset`, or `global`. The best I could do would be to translate a Tcl `set` command into a sequence of several Java instructions that did the appropriate checks. Unfortunately, the number of Java instructions to implement each Tcl command would make the compiled program too big. A more realistic scheme is to generate Java bytecodes that call

one or more Java methods to do the actual work for each Tcl command. With this number of Java method calls, acceptable performance would depend on using a Java machine code compiler. But these compilers won't be free.

Another problem is that much of the interesting code in Tcl/Tk and its extensions is in C. Java code can call "native" methods implemented in C, and vice-versa, but this is awkward and the capability is disabled in Netscape (and probably most other Java implementations) for safety reasons.

11 Implications for current script and extension writers

11.1 Implications for scripts

Use lists. They are now even faster than arrays since indexing elements requires no hashtable lookup.

You should not rely on the string representations of lists having a particular syntax. That is, you should use list operations like `lindex` to manipulate lists. Also, list operations will now parse the entire list when converting an object to a list. `lappend`, for example, no longer ignores arbitrary text in the list it is appending an element to. This means that you shouldn't use list operations to manipulate values that aren't lists. Use string operations to manipulate arbitrary strings.

Use braces around expressions, including those used in control structure commands. This lets us generate inline instructions to evaluate the expression without the need to check for second-level substitutions that might invalidate the code.

Put all `global` commands at the start of procedures.

The execution traceback information in error messages will change. Since the compiler will generate inline instructions for what currently are recursive calls to `Tcl_Eval`, error tracebacks will be somewhat flattened. They should be more understandable, however, especially since I should be able to include source line numbers.

11.2 Implications for extension C code

Convert string-based command procedures to use objects. These will execute faster and will be able to take advantage of type-specific operations that operate on internal representations appropriate for those types.

As described above, don't use the list API procedures to operate on values that aren't lists and don't rely on them preserving white space between list items.

12 Future work

A substantial amount of work remains on the compiler. This includes the changes described above, in particular:

- Compile inline code for `expr` commands.
- Implement the remaining changes for Tcl variables. This includes compiler-allocated entries for global variables, and better `global`, `unset`, `uplevel` and `upvar` support.
- Generate inline instructions for high payoff commands. For example, I won't immediately compile the `clock` or `history` commands.
- Add namespace support.

I expect to release an initial version of the bytecode compiler about four months from now.

I have no plans at this time to do type inference for Tcl expressions as done by David Koski [Koski95] and Guy Steele [Steele94]. This can be very effective—David Koski got speedups of more than a 1000 for some floating point intensive Tcl code—but type inference is difficult to do correctly in a language as dynamic as Tcl.

13 Conclusion

I have described the design and current state of an on-the-fly bytecode compiler for Tcl. I expect this compiler to eventually improve the speed of compute-intensive Tcl scripts by a factor of about 10. Part of the compiler's speedup derives from its use and support for dual-ported objects. Early results with the compiler are promising but considerable work remains.

14 References

[Howlett94] Howlett, George. "Packages: Adding Namespaces to Tcl." Proceedings of the 1994 Tcl/Tk Workshop, New Orleans, Louisiana, June 1994.

[Koski95] Koski, David. "A Tcl Compiler." Unpublished class project report, University of Wisconsin, October 1995.

[McLennan95] McLennan, Michael. "The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More." Proceedings of the 1995 Tcl/Tk Workshop, Toronto, Canada, July 1995. Also described by the web pages at <http://www.tcltk.com/itcl/namespace.html>.

[Rouse95] Rouse, Forest R. and Christopher, Wayne. "A Tcl to C Compiler." Proceedings of the 1995 Tcl/Tk Workshop, Toronto, Canada, July 1995. A commercial version is described by <http://icemcfd.com/tcl/ice.html>.

[Sah94] Sah, Adam. "TC: An Efficient Implementation of the Tcl Language." Master's Thesis, UC Berkeley Dept. of Computer Science report UCB-CSD-94-812. 1994.

[Steele94] Steele, Guy. Unpublished Common Lisp program that translates a subset of Tcl to C.

[Welch95] Welch, Brent. "Customization and Flexibility in the exmh Mail User Interface." Proceedings of the 1995 Tcl/Tk Workshop, Toronto, Canada, July 1995.

Tcl/Tk as an OpenDoc Scripting Part

Jim Ingham
AT&T Bell Laboratories
jingham@mhcnnet.att.com

Abstract

This paper describes the advantages that would accrue both to Tcl/Tk and to OpenDoc by importing the Tcl/Tk scripting system into the OpenDoc environment. First, it gives a brief outline of the OpenDoc compound document architecture. Then it describes the place Tcl/Tk could fill in the OpenDoc scheme. Finally, it gives a preliminary sketch of the areas in which Tcl/Tk would need to be modified in order to implement this merger.

Introduction

The OpenDoc component document architecture¹⁻⁶ promises to revolutionize how modern applications are deployed to users. Backed by the Component Integration Laboratories (including Apple, IBM and Novell), it promises to cure the bloat of modern applications, which feel the need to provide every feature possible within one shell. More important, OpenDoc provides unique opportunities for smaller developers, since they can concentrate on solutions in their particular area of expertise, without having to provide the whole environment within which their solutions can function.

The basic idea of OpenDoc is to cast the modules that comprise an ordinary application: text editors, spell checkers, picture editors, a differential equation solver, an FTP channel ..., as separate "parts" that can be embedded in arbitrary combinations into a document shell. The document shell, provided by OpenDoc, controls the event loop and storage, arbitrates access to system resources like the menubar, mouse and keyboard, and provides a channel of communication among the parts for geometry management and data passing.

OpenDoc provides a rich language by which the parts communicate, but it does not come with many prebuilt features. While the geometry arbitration mechanism is well defined, the developer is free to choose the particular model of geometry management according to his or her needs. OpenDoc uses the Open Scripting Architecture (OSA)^{1,7-8} to facilitate data passing among the parts, but does not provide any particular scripting language. Nor does it come with a toolkit for implementing ordinary GUI functions (buttons, listboxes,...).

This situation provides an interesting opportunity for deploying Tcl/Tk in a new context, which has many attractive features. If we could make Tk a "part" in the OpenDoc world, then the creators of OpenDoc content would get the ease of development and the rich widget set for their documents that has been such a boon in the X world. At the same time, Tk developers would gain the ability to embed whatever parts are available in the OpenDoc world into their application, without having to build extensions for those parts. This kind of embedding could fill the role that the complex VBX's fill in the Visual Basic world.

Also, since OpenDoc is intended to be a cross-platform architecture, the 7.5/4.1 versions of Tcl/Tk are a natural choice for the scripting architecture/widget set. In the rest of this paper, I will give an outline of how Tcl/Tk could be used as a tool in the OpenDoc environment. First I will provide a brief summary of OpenDoc. Then I will sketch the enhancements that need to be made to Tcl/Tk in order to fulfill this role. There are several levels of OpenDoc incorporation, and while the tightest coupling would require some rather low-level changes to the Tk core, there is an intermediate level that could be done fairly surgically. I have started the work in some of these areas, mostly at this point as an assay of the level of difficulty.

Although there is much work to be done, there do not seem to be any show-stopping difficulties, and results are promising enough to make the effort worthwhile. This paper is really meant to be a call for an interest group. I intend to show why this is an interesting project and sketch its parts, in the hope that I can find some other developers who are interested in cooperating in its implementation. Note that the following discussion is in the context of the MacOS port of Tcl/Tk. There are also ports of OpenDoc to AIX and OS/2 currently available, with a Windows port due in the middle of the year⁵, but I have no experience with these ports.

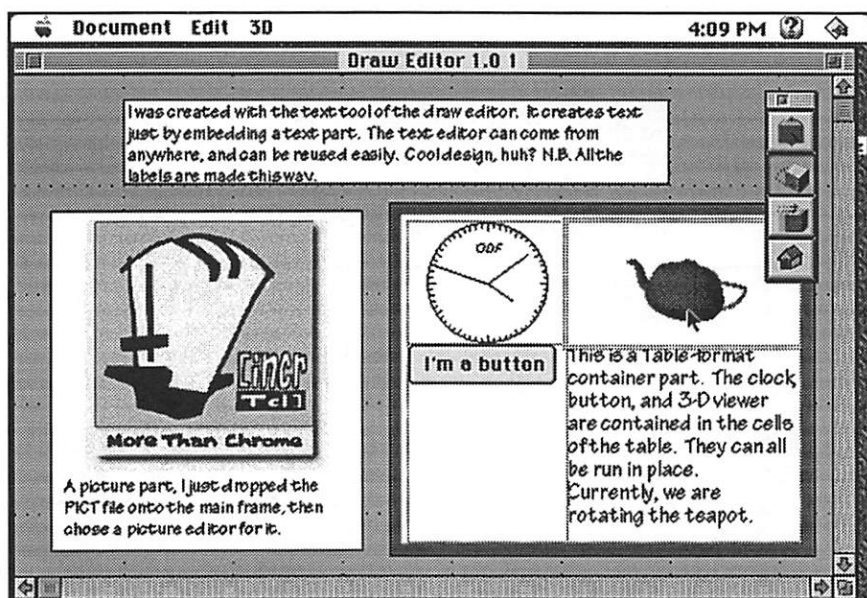


Figure 1. Here is an example of an OpenDoc document. The root part is a simple drawing editor. The Toaster picture is an embedded PICT, which is controlled by a Picture editor that comes with OpenDoc. The Clock, teapot and button are all embedded in a Table part. The teapot part, which is currently active, is a QuickDraw 3D document. This display was created with parts from Apple's OpenDoc Developer Release 5 CD.

Summary of OpenDoc

The fundamental idea of the OpenDoc model is that the document, rather than the application, should be the central focus of attention. The motivation is to give the content provider control over what elements go into each document: If I don't need pictures, then I don't need the picture editor most word processors now routinely include... And if I need a picture editor, or some other special service embedded in a text

document, I should be able to choose one made by someone whose primary interest it was to provide that function, rather than the knock-off required to meet some feature spec sheet.

Each OpenDoc document is broken down into a number of "parts," each of which is controlled by its own "part editor." A part editor could be a text or pic-

ture editing facility, a spell checking facility, a TCP/IP socket interface... In a given document, there may be several parts that are controlled by the same part editor. By implementation, the part editors are actually class definitions, derived from a base class provided by OpenDoc, and the parts are instances of their part editor class. So although the parts owned by the same part editor share code, they are really independent entities, with their own data and state.

The user does not manipulate part editors directly. Rather, the developer of the OpenDoc part also makes a "stationery" document for the part editor. This can be as simple as a template for one part, or that part with some state information included, or can be a whole conglomeration of parts. The user opens the stationery document, which automatically makes a copy of itself for the user to edit. The stationery documents also support a drag and drop mechanism. So if you want to add a picture to a text document you are editing, you drag the stationery document for the picture part editor from the Finder onto your text document, and drop it wherever you want the picture to go, then interact with the picture part to load in the desired image. Thus you can make a complicated collection of parts, save it as a stationery document, and it becomes an atomic unit that you can later insert into another document.

Each part usually controls some screen area. For instance, a text part would have some area in its document's window to display its contents. Each contiguous bit of screen area in a window that a part occupies is called a "frame." Each part can have more than one frame, and the frames owned by a given part need not be in the same window. So, for instance, if I wanted to put a long bit of text into a newsletter document, I could break the text's part up into a number of frames, and spread the frames throughout the newsletter. There is no requirement that the frames be rectangular, though this has little implication for our purposes. More importantly, the frames of a given part can be constructed in such a way as to contain other parts.

This containing relationship can be used to solve long-standing problems, like inserting a picture into a text document. One frame of a text part could contain a picture part which is in charge of displaying the image. The containing frame (the text editor in our case) manages the geometry of the embedded part (the picture editor), but has no other responsibility for it. When redrawing is necessary, OpenDoc forwards update requests to all the embedded parts, and each part has the responsibility for drawing itself. Thus the coupling between an embedded part and its container is minimal. The embedding can go on to any depth, so a cell of a spreadsheet can contain a text part that contains an embedded picture, and so on.

To maintain consistency in this heterogeneous environment, all user interaction is routed through the OpenDoc shell. For each system service — the mouse, the keyboard, the menubar and the systems ports — there is an associated "focus." Parts negotiate, through the OpenDoc shell, for the focus for a given input stream. In general this is done by "activating" one part, which then requests the keyboard and mouse focus. There are user-interface conventions for how the active part is displayed, and how the activation is switched from one part to another (this is done by a single mouse click in the part to be activated, for instance). Once a part gets the focus for a given input, the OpenDoc shell will route all such events to that part.

Finally, the OpenDoc shell provides the facility for one part to send scripting events to another part in the document, using Apple's Open Scripting Architecture (OSA). The OSA provides a well defined way for parts to publish their interface to other parts, and to OSA aware editors. To achieve this communication, the parts build up "Apple Events," which describe a set of steps to be taken using this interface, and pass them back and forth. Any part that is OSA compliant, i.e. that has this interface, can take part in these scripting exchanges.

Tcl/Tk in OpenDoc

Let's start with a scenario of how Tcl/Tk could be used in OpenDoc. Suppose I have written a 2-D differential equation solver, which I have incorporated as a Tcl extension so that I can script it. Then I wrote a nice Tk front-end to control the simulation. Now I need to display the data. Since it is not entirely trivial to write a good 3-D data viewer, particularly one that can do convincing coloring and shading, rotations, etc., and perhaps produce a QuickTime movie of the results, I will look for a commercial package to do the task.

All this is possible in the context of ordinary applications, however, the presentation is fragmented. I start up my Wish, do the simulation, and then start up the 3-D viewer, play with it a bit, then go back to the Wish to redo the simulation. I have to have a data file somewhere to store the input cards, and the results somewhere else, and my movies in yet another file...

If both Tk and my 3-D viewer were OpenDoc parts, however, I could make up a single Document that contains as its parts, the Solver engine, the GUI and the viewer window, sort of a poor-man's Mathematica. Later, if I wanted to run a series of simulations, I could add a spreadsheet part to store a table of test cases, write a Tcl script to get the columns one by one from the spreadsheet, run the simulation for each column, and write back some results to the spreadsheet. I could even embed the QuickTime movies of the results as icons in the spreadsheet, if the spreadsheet was written to contain other parts in its cells. Finally, my Tcl script could drive the spreadsheet to do some analysis of the results.

All this would be within a single document, so I can hand it off to a colleague to use, without having to give complicated instructions on its use (for instance describing the naming convention that links the input and output data sets). Furthermore, I can make copies of the document, each containing a separate test sequence, and each copy is self-contained, with all its data and results in one place.

This picture relies on two separate facilities: the intercommunication of the parts which makes joint operations possible, and the common shell by which a

single environment can be constructed around all these functions, some of which I control (the Tcl/Tk and solver engine), and some of which I don't even have libraries for (the spreadsheet and 3-D viewer). I will discuss in the next section what needs to be done to provide these two facilities.

The advantage of this integration goes in two directions. Tk brings to the OpenDoc world an extensible (I could incorporate my 2-D solver) scripting language, with a well-designed widget set. In return, Tcl/Tk applications will gain access, in a particularly seamless and attractive way, to custom facilities that are not available as libraries for incorporation into the Tcl shell.

The whole discussion is speculative at present, since quality applications available as OpenDoc parts are just coming on the scene. Cyberdog⁹, Apple's OpenDoc based Internet Environment (with integrated e-mail, internet news service and WWW browser) shows great promise. It has some nice primitive widgets, like embeddable button parts that are linked to a URL, and on activation raise the appropriate browser window for that URL.

But this will likely change in the near future; OpenDoc won Best Technological Achievement of 1995 at Comdex¹⁰, and has been consistently rated superior to the only real competitor, Microsoft's OLE (see ref. 2 for an exhaustive discussion of the merits of OpenDoc over OLE). It was chosen by the Object Management Group as the standard desktop service for the CORBA environment¹¹. The OMG is an industry group responsible for developing standards for object technologies.

There is an impressive list of commercial developers committed to writing OpenDoc parts (see <http://www.cilabs.org> for details). Claris announced that the next version of ClarisWorks will be an OpenDoc container part. Netscape Communications announced at this year's WWDC conference that it will provide some support for OpenDoc, but the details were sketchy. And at the same time, Tcl/Tk has an opportunity to make a contribution to this development precisely because there are no well established competitors.

Changes to Tcl/Tk

This section falls under two heads, OpenDoc compliance, and OSA compliance. It is worth pointing out that these two are independent tasks. In fact, the OSA compliance should be implemented regardless of whether the OpenDoc work is attempted at all. It is the logical way to implement exec and perhaps send on the Mac. Both the Tcl-based Mac Editor Alpha¹² and the Mac implementation of Perl offer Apple Event support. In the case of Alpha, this has made it a very powerful plug-in editor for the major Mac development environments. Without some kind of Apple Event support, Tcl/Tk will be a mute and isolated participant in the MacOS world. I have made a start at implementing the OSA interface to Tcl/Tk, and will have a beta release of this by the time of the conference.

OpenDoc:

There are two levels of OpenDoc incorporation offered by the Apple OpenDoc development group. One is to make Tcl/Tk a full fledged OpenDoc part, capable of being embedded in other parts, and of containing embedded parts. This is called a container part, in OpenDoc terminology. It is obviously the best way to proceed, since it would allow us to embed Tk controls in any other document, and to embed viewers, spreadsheets or whatever parts we wish within the Tk layout of our application. However, it also takes the most work, some of which is not entirely trivial.

The second is to make Tcl/Tk be an embeddable part, but not a containing part. This would allow us to put Tk controls and display widgets into a document alongside other parts, and use the Tcl interpreter to run our own extensions, and communicate with the other parts. However, we would not, for example, be able to embed the 3-D viewer into a Tk text widget, or use the Tk scrollbar widget to drive a non-Tk part.

NECESSARY ENHANCEMENTS: The full OpenDoc support requires changes or enhancements in four major areas, the event loop, windows, part activation, and part storage. The following is a bare sketch of each of these four areas.

The Event Loop: In any compound architecture like OpenDoc, the shell has to control the event loop. In

order to switch control between the various parts, and also allow the parts to share idle time, there has to be a central router. In OpenDoc, each part has a `HandleEvent` method that does the actual business of responding to a single event. The document shell takes each event off the event queue, determines which part should respond to that event, and dispatches the event to that part's `HandleEvent` method.

The Tk event loop would have to be replaced by an event handler, which can do what it wishes with the events in Tk's internal event queue, but cannot poll for system events. This is not too difficult to implement. The current model in Mac Tk is that each time through the event loop, the top of Tcl's internal event queue is serviced. If there was an event in the queue, it is handled and the event loop is restarted. If the queue is empty, then the various event sources (file events, as well as UI events,...) are all polled for events. The new events are not processed at this time, but just added to Tcl's internal queue. Then the top event in the queue is serviced.

In OpenDoc, the handler would just put the event that woke it up onto the queue, then service the top event. The polling for file events, etc.. would be done when the handler was invoked for an idle event. It is yet to be seen whether this will cause performance problems, but it is the model for all the OpenDoc applications, and none of the currently available parts seem at all sluggish, so it should not be a problem.

Windows: One major change in Tk required by OpenDoc is that the main window of the application may not be a top level window. After all, we want to be able to embed a Tk layout into a document. This does not seem a very hard problem, however. We can just make an `ODToplevel` widget, which is just like a `TopLevel`, except that its window manager is the OpenDoc shell rather than the MacOS window manager. Tk would draw all its component widgets into this `ODToplevel` just as it would into an ordinary `toplevel`. Then it passes its final geometry request to its containing part.

There is some work to be done to implement the "wm" and "winfo" commands for these `ODToplevels`, but OpenDoc possesses a full set of calls to get geometry information, and arbitrate for new geometry, so the groundwork is well laid. One requirement for the

ODToplevel, and ordinary Toplevels, is that they must be registered with OpenDoc. This is so OpenDoc can do event routing, and dispatch update requests. This registration is accomplished by a fairly simple call, though we have to get access to our parts object to make it. Thus windows in OpenDoc Tk must have a part pointer added to their data storage. This can just be a ClientData pointer hung off the Tk_Window structure, so it is not too intrusive.

If we want to implement Tk as a containing part, we will also have to provide an ODFrame widget. This will be the containing frame for an embedded OpenDoc part. Each OpenDoc part is responsible for informing its containing part of its geometry requirements, so the ODFrame has all the information it needs to negotiate for position within the rest of the Tk layout. Further, each OpenDoc part has the responsibility to draw itself, so we don't need to do any work to present the contained part.

There are some User Interface guidelines which indicate which part is the active one in the document; the ODFrame widget would need some custom border drawing routines to conform to these requirements. Since OpenDoc does all the event passing, the ODFrame's responsibilities to its contained parts are limited to activating them when the frame is clicked in, and redrawing the borders in accordance with the UI Guidelines when the contained part becomes inactive.

Part Activation: When the Tk part is activated (for instance, when it is clicked in, or when it is first created) OpenDoc requires a number of setup routines be run. The part has to request focus for the events it is interested in (mouse events, keyboard events, as well as other system functions, like communications ports). This is done through set of fairly simple library calls, and will be easy to do. The UI requirements for displaying the active part (there is a little double line border in the current implementation) must be implemented. This frame would just be a special -relief option for the ODToplevel.

Again, when the part is deactivated, there is a protocol that must be followed to resign the foci which the newly activated part requests. The general UI standard for OpenDoc is that a part withdraw its floating palettes when it becomes inactive, and redraws them when it is reactivated. This would also have to be implemented.

Part Storage: Each part is responsible for reading itself in from storage, and writing itself out to storage. This is so the document can be saved and restored to this state later. In Tk, this is relatively easy, since the Tcl/Tk architecture allows you to query out the state of the interpreter. You know what all the global variables are, and you can just run the Tk tree to find out all the widgets, their contents and their positions.

You also need to track the proc definitions, what files are open, what fileevent handlers have been registered, etc. The state of the application can be "pickled" as this set of Tcl/Tk commands which restore this state. Doing this quickly and efficiently may take a little work, but all the facilities are already available in Tk. Of course, any extensions that you add to Tcl will have to have this ability to pickle themselves as well, if they are to coexist in OpenDoc.

Open Scripting Architecture:

This is the generic language by which parts in OpenDoc (and applications under MacOS) communicate with each other. The idea is that each part specifies the objects and operations that it wants to export to the world. For instance a generic part has objects like documents and menus. A word processor has, in addition, paragraphs, lines and words of text. These objects have various properties, like the entries in a menu, or the font of a character. There are also actions that can be performed on these objects, like setting the font of a word object, hiding a window, or telling an object to run one of its methods.

These verbs and objects are combined into "suites," many of which are standardized by Apple. There is a "Required suite" of operations that a generic application must perform: open, open document, print, quit and run. There is the "Core suite" that supports basic operations like get, set, delete and save. There is a "Text suite" that adds objects like characters, words, and paragraphs. You can support all of a suite, or any part of it that is appropriate. And of course, any part is free to add its own suites as well.

Each part (or program) has a Dictionary resource (the 'aete' resource). In this resource, each of the program objects and verbs is specified both by an internal four-letter code and by some "Human readable" information on the object. Another application can read this resource, and determine the scriptable objects in your application. An example of this is given

in Fig. 2. In this figure, Apple's Script Editor application has read in the dictionary, and presents the human readable component in the window at the lower left.

At startup the object and verb codes are linked to handlers within the part. When another part or application wants to send an instruction to the part, it constructs a data structure describing the operation and the objects it acts upon (called an "Apple Event"). This is a nested structure, so that objects

the target application responds to the event by parsing it up into its parts, resolving all the object references, calling out the handlers associated with the verbs, and finally returning a reply Apple event, if the incoming instruction requested a result.

For our purposes, we do not need to provide an interface to all the facilities within Tcl/Tk, since it already has its own scripting language. If a part wants to build up a Tk GUI layout, the best way would be to create a Tk part instance and send it the necessary

Tcl code. This is already possible in Tk4.1, using the standard "do script" event.

Note that the OSA provides another, even nicer, way to run Tcl scripts. You can register the Tcl interpreter as a "Scripting Component." A scripting component contains the interpreter as a shared library. Any part can ask the component manager for an instance of the interpreter, and then feed commands to it. For more details, see chapter 10 of ref. 7. This method is useful because you do not have to start up an external wish part to run your scripts: a part could contain the Tk component internally. It is not appropriate, however, if you want the Tk GUI to drive multiple parts.

Vince Darley has made a start on this approach for the MacOS by wrapping Tcl as a scripting component. With this, you can use any of the standard script editors (for instance Apple's Script Editor) to write your Tcl

code. However, before we can do the same for Tk within OpenDoc, we have to separate out the Tk event loop as I discussed in the previous section.

When designing an OSA model for Tk, we should concentrate on the things a client part will really want to be able to do. This consists mainly of allowing other parts to activate elements of the Tk GUI, to access and modify data in the text, entry, listbox and canvas widgets, and perhaps to modify configuration options of the widgets. I have made a start on this, as is

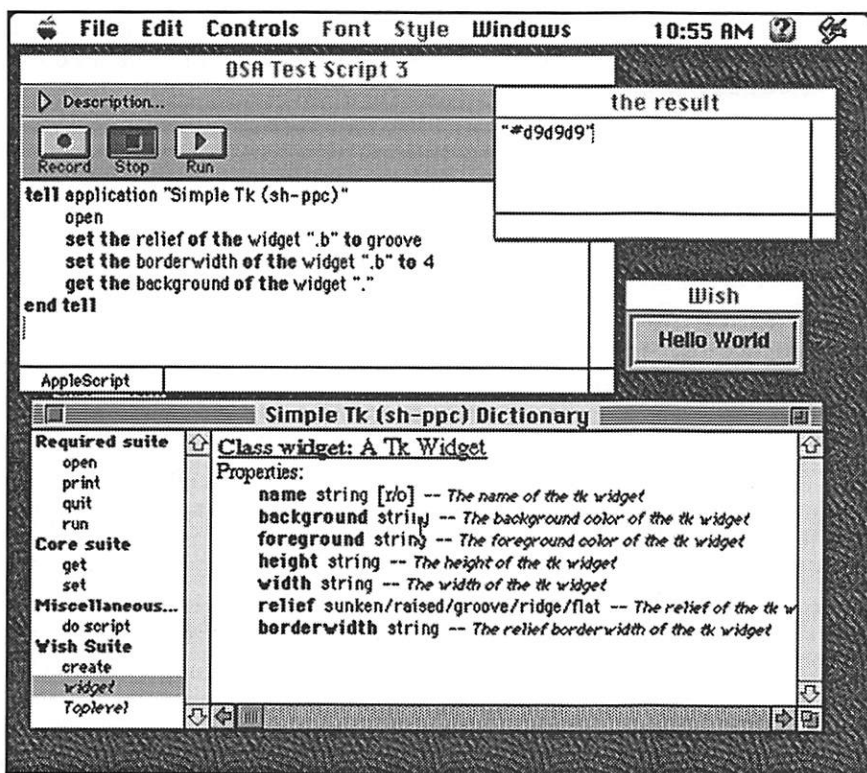


Figure 2. This is an example of an OSA model for Tk. There is a widget class that encompasses the basic widget behavior. Each of the widgets are derived from widget. The configuration options are accessed through the get and set verbs. The widget commands will be accessed by the "tell" verb. This is my current focus of interest, and I will release a working beta soon.

can be specified in quite complicated ways. For instance a complicated object specification like "the first paragraph in the window "My Article" whose font is "Courier"" can be quite easily built up.

Then it passes this Apple Event to the Apple Event Manager, which routes it to the target application or part as a "High Level" event. In OpenDoc, the shell passes the event to the parts HandleEvent method. A MacOS application receives the instruction as an entry in its event queue. In either case,

shown in Fig 2. I have implemented an OSA extension which can get and set configuration options, and, using the AppleScript standard "tell" construct, activate widget commands. The next job is to apply the objects in the OSA "text suite" to the text, listbox and entry widgets. I am currently working on this.

It would also be very useful for Tk to have some facility to build up an Apple Event and dispatch it to another application. The exec command is not currently implemented in the Mac port of Tk. This is because MacOS does not implement a Unix style exec. The Apple event mechanism fulfills this role in the MacOS. In fact, it is much more powerful than exec, since it also fulfills a role like the Tk "send" command. The dictionary mechanism is a very nice touch for documenting the interface.

The capability to build Apple Events already exists in the Tcl-based Mac Editor Alpha. It has a Tcl extension that allows you to build up arbitrary Apple

Events from the object and verb codes, and send them. Peter Keleher has made the code available to Mark Roseman. It is based on a simple wrapping of a library (AEGizmos) provided by Apple, which is in the public domain, so we can very easily add it to Tk.

Another example of the use of this Apple Event builder is to implement "send" for Tk on the Mac. After all, "send" is no more than the doScript Apple event with the script as its data, sent between two Wish applications. This is not the best way to implement "send," of course, since it means that the command has very different implementations under X-Windows and the MacOS. But it would be a nice quick way to get this functionality. It could fill the need till a truly cross platform solution is devised.

Conclusions

First of all, I hope I have communicated some of the promise of the "document centered" User Interface philosophy. It allows the user to tailor "mini-applications," which contain the specific functionality they need. Moreover, each copy of the document contains both the description of the "application" that the user has assembled, and its state when he or she last left it. So, once constructed, the document can be replicated, reused, edited, and embedded as an atomic element in another document. OpenDoc transfers many of the "application designer" functions to the user of the application, and offers a flexibility and level of control that is quite promising.

However, at this stage the OpenDoc world is lacking a good solid GUI toolkit. This, by the way, is not some oversight on the part of the OpenDoc development group. The idea was to create an open architecture that developers could tailor to their own needs, rather than forcing a whole package down everyone's throats. This situation offers great possibilities, however, for someone who can deploy a solid GUI toolkit in OpenDoc. There are several reasons to think Tcl/Tk would fill this role elegantly.

Its extensibility would be a great asset, just as it is for stand-alone applications. The widget set has proved

popular because it is easy to use, and yet very flexible. Currently OpenDoc only has a solid implementation under MacOS. However, as it is deployed to more platforms, we can repeat the steps we took to embed Tcl/Tk into MacOS OpenDoc, with the confidence that the engine is well tested on those platforms. This would not be true of any widget set that was developed primarily on the Mac.

I will conclude by describing the steps required for the incorporation of Tcl/Tk into OpenDoc. First of all, we need to refit the event loop part of Tcl, converting it into a handler based architecture. I made a quick test of this with one of the early 4.1 beta's, and there were no substantial impediments. I have been waiting for Tk to come out of beta before attempting the job in earnest.

The second crucial step is write an OSA interface to Tk. This can be done as a true extension, without modifying the Tcl/Tk core. The results will be of sufficient utility to MacTk in general, that the work should be done as soon as possible. I have made substantial progress on this task, producing a version that implements the basic widget objects. It can query and set the configuration options, and run widget commands. I will make a first draft of the inter-

face to the Text widget, and then release what I have done. Look to Mark Roseman's Mac resources page¹³ for the results.

The next step is to embed Tk into OpenDoc. The simplest way to do this is to start by allowing only separate, Tk-controlled toplevels. These only need to be registered with OpenDoc, so that they receive events; other than that they do not differ from their Wish counterparts. This was easy in the early beta, and should suffice to get a test version running. Then we need to write an ODTopLevel widget, and finally the ODFrame widget.

Since I did my early experiments, the OpenDoc Development Framework¹⁴ has also come out of beta. This is a framework from Apple Computer for writing OpenDoc parts. It takes care of many of the Human Interface details, and isolates the system specific parts of an OpenDoc part. It looks very promising, and can speed the writing of OpenDoc parts. I will use this framework to implement the TK embedding.

References

- [1] Apple Computer Inc.
OpenDoc Programmers Guide For The MacOS
Osborne McGraw Hill Press, December 1995
- [2] R. Orfali, D. Harkey & J. Edwards
The Essential Distributed Objects Survival Guide
John Wiley & Sons, 1996
- [3] A. MacBride & J. Susser
Byte Guide To OpenDoc
Osborne McGraw Hill Press, 1996
- [4] Apple Computer OpenDoc Home Page:
<http://www.opendoc.apple.com>
You can get OpenDoc, the developer's kit, Apple's OpenDoc White Paper, and many other goodies from this site.
- [5] IBM's Club OpenDoc Home Page:
<http://www.software.ibm.com/clubopendoc/index.html>
IBM is responsible for the OS/2, AIX and Windows Ports of OpenDoc. You can find information related to these ports, and other OpenDoc goodies here.
- [6] CILabs Home Page:
<http://www.cilabs.org>
CILabs is the industry group in charge of coordinating the standards for OpenDoc. They help organize the interpart communications, and set up classifications for various part types (spreadsheets, text editors, database access, ...)
- [7] Apple Computer Inc.
Inside Macintosh: Interapplication Communication
Addison Wesley Publishing Company, 1993
- [8] Dave Mark
The Ultimate Mac Programming Guide
IDG Books Worldwide, 1994
The first section of this book is about implementing OSA in a traditional Mac application. It also contains several useful reprints from Develop Magazine in the appendices.
- [9] The Cyberdog Home Page
<http://cyberdog.apple.com>
Here you can download Cyberdog, read the Cyberdog white paper, and soon get the SDK for Cyberdog as well.
- [10] The press release is at:
<http://product.info.apple.com/pr/press.releases/1996/q2/960206.pr.rel.opendoc.html>
- [11] The press release can be found at:
<http://www.cilabs.org/pr/pr.omg.html>
- [12] The Alpha Home Page is:
<http://www.cs.umd.edu/~keleher/alpha.html>
- [13] Mark Roseman's MacTcl resources web page is:
<http://www.cpsc.ucalgary.ca:80/~roseman/mactcl/>
- [14] The OpenDoc Development Framework Home Page:
<http://www.odf.apple.com>

In Search of the Perfect Mega-widget

Stephen A. Uhler

Sun Microsystems Laboratories

ABSTRACT

Mega-widgets are Tk widgets that can be created entirely in Tcl, yet behave indistinguishably from their native counterparts. Although several different frameworks have been constructed to create and manage mega-widgets, none does a perfect job, as there are aspects of a native widget's behavior that can't be simulated strictly in Tcl. We propose a small set of enhancements to the Tk event binding and focus models that will enable mega-widget frameworks, including those that are currently available, to do a better job supporting the semantics of native widgets. This allows a script that uses a native widget on one platform, to use a mega-widget implementation of the same functionality on a different platform without the need to know whether the widget is native or not.

Introduction

One of the primary reasons for Tk's success is the ease in which user interfaces can be constructed, by gluing together one or more of the 13 native user interface elements, called widgets, in Tcl. However just as it is easy to use the native widgets, it has proven difficult to create new ones. They need to be written in C and conform to a fairly complex set of behaviors. As a result, each new user interface ends up being constructed out of the same basic building blocks every time. Even though common combinations of widgets can be manipulated as a unit by Tcl procedures (like `tk_dialog`), such procedures are hard to share, as their procedural interfaces tend to be different than the object based paradigms used to manipulate the native widgets.

Mega-widgets

The sharing and reuse of common combinations of widgets can be greatly enhanced if they are packaged to look and behave exactly like the native widgets. These widget combinations that act like native widgets were dubbed mega-widgets by Mike McKlenan, who implemented them for his `[incr Tk]`¹ mega-widget framework that is based upon `[incr tcl]`.²

There have been other frameworks constructed for building mega-widgets, such as Tix by Ioi Lam,³ and `[Incr Widgets]` by Mark Ulferts.⁴

Tk now runs on several different computing platforms that use a different look and feel for common user interface actions. As Tk moves to support the native widgets on each of its platforms, programmers will need to program to higher level interfaces than provided by the basic widgets, to insure that

scripts not only run unchanged on various platforms, but conform to the native look and feel as well.⁵

Where useful native widgets are only available on some platforms they will need to be created on those platforms that don't provide them. Most likely these widgets will be created as mega-widgets, rather than the more difficult and arduous task of building them in C. As a consequence, a script that uses a native widget on one platform, might find itself using a mega-widget on another, completely unbeknownst to the program's author or user.

In the past it might have been acceptable for mega-widgets to behave almost like native widgets, whereas in the cross platform environment it is essential that they behave exactly like native widgets, as script writers might not have control over which widgets are native, and which ones are constructed as mega-widgets.

The Missing Pieces

Overall, the existing mega-widget frameworks do a good job providing users with the illusion that they are dealing with native widgets. The frameworks support the standard widget commands, such as `cgets` and `configure`, and handle configuration options in the expected way. Shannon Jaegar⁶ suggests the features that a mega-widget framework should provide, as well as compare the various features of several existing frameworks.

In order to pass the litmus test of an ideal mega-widget, the user of that widget (the application programmer) must not be required to know whether it is native or not. Although it is acceptable for the programmer to ask about the inner workings of a mega-widget if they choose, assuming the mega-widget framework permits it, it is not acceptable to force the user to know about information that would be hidden if the widget was native. There are two areas where the frameworks fall

short: event bindings and focus behavior.

The primary failing of the existing mega-widget frameworks is their inability to permit users to associate event bindings with a mega-widget without compromising the illusion of the widget as an atomic entity. Consider the following example:

```
some_widget .widget
bind .widget <ButtonPress-1> {
    puts "%W %x,%y"
}
```

If `some_widget` was a native widget, then no matter where inside the boundaries of `.widget` the user pressed the mouse button, the value of `%W` would be `.widget`, and `%x` and `%y` would be pixel offsets from the top left corner of `.widget`. On the other hand, if `some_widget` was a mega-widget, comprised of many sub-parts within its border, the same binding would likely report `.widget.sub-part` for `%W` where *sub-part* is the name of one of the mega-widget components, and `%x` and `%y` would no longer be relative to the `.widget` container, but instead be offsets from `.widget.subpart`. Thus the user is forced to deal with the details of the mega-widget's components, which they shouldn't need to know anything about.

Similarly, if the user issues the Tk `focus` command when the focus is inside a component of a mega-widget, `focus` would return `.widget.sub_part`, again exposing the user to information about the inner workings of the widget that they shouldn't be required to know about.

Both of these problems result from Tk's notion that there is a one-to-one relationship between a widget and the underlying C window object for that platform. The events and focus information come from the underlying window object, not the widget command. In a mega-widget, there can be multiple underlying window objects for each widget instance, which causes Tk to report information about

hidden parts of the mega-widget.

A Solution

Tk needs to be extended to correct for the discrepancies in the event bindings and focus command when using mega-widgets. These extensions should allow the existing mega-widget frameworks to work better without major changes, and do so in an architecturally neutral way that doesn't favor one style of mega-widget framework over another.

We have modified the `frame` command to accept another configuration parameter, `-command` to provide a convenient hook for attaching Tcl code to a widget command. Like the current `-class` option, `-command` may only be set at widget creation time, and not modified using `configure`. When the `command` option is specified, it changes the behavior of the frame, making it a mega-widget container. Calls to the created widget run the tcl code specified as the `-command` value, with any arguments concatenated. A new frame subcommand, `really` causes any additional arguments to be passed to the actual frame widget, and not be diverted to the `-command` string. For example, to create an instance of the *mega* widget called `.mega-widget`, the mega-widget framework would issue the following commands:

```
frame .mega-widget \
  -class Mega \
  -command {mega .mega-widget}

proc mega {name args} {
  $name really configure...
  # other processing...
}
```

The procedure `mega` would be called to actually process the arguments of the `.mega-widget` command, which would implement the behavior of the widget in Tcl code. The `-class` option could be used to initialize the default bindings for the *mega* widget. The `mega` procedure can access its containing

frame using the `.mega-widget really` option.

In addition to diverting the widget processing to tcl code, the `-command` option turns on special behavior for event bindings and focus, so that keyboard and mouse events that arrive for any children of `.mega-widget` will be reissued as if they occurred for `.mega-widget` directly. This process is repeated for each enclosing mega-widget container, supporting arbitrary levels of mega-widget nesting. Thus if the user sets a mouse binding to a mega-widget that happens to fire while the pointer is over one of the widget's sub-windows, the event will fire once for the sub-window, to process any bindings that were set on the sub-window by the mega-widget builder. Then the event will be reformulated as if the sub-window wasn't there, and trigger again, executing the user's binding set on the container. The user sees the mega-widget as an atomic entity - just like a native widget, yet the mega-widget creator can set and process events on the sub-windows as needed.

As an aid to mega-widget builders and debuggers, an new subcommand of `wininfo` has been added:

`wininfo container window_pathname.`

This command returns the empty string if the specified window is not in a container. If the window is a container, `window_pathname` is returned. Finally, if `window_pathname` is a sub-component of a mega-widget, then the name of its nearest enclosing container is returned.

The final modification to Tk is in the `focus` command. The `focus` options that return window names (e.g. `focus` and `focus -lastfor`) now report the outermost container for the widget that has the focus. This brings the behavior of mega-widgets in line with native widgets. `Focus` will not report a sub-window name that should be hidden from the user. The new command:

`focus -container container_name`

can be used to determine the actual sub-window within a mega-widget that has the focus, by returning the outermost container (or native widget) that is inside *container_name*.

Implementation considerations

This mega-widget extension is implemented by adding two new flags to the Tk window structure, an *is_container* flag and an *is_contained_in* flag. The event rewriting and propagation only occurs if the *is_contained_in* flag is set, so there is no performance penalty if mega-widgets are not used. The entire extension is about 200 lines of C code which implements five functions: the recursive binding dispatch, the event rewriting, the new frame command option, changes to the `focus` command, and the implementation of `wininfo container`.

Two C interfaces are provided to set and query the new state bits.

Summary and conclusions

We have made small changes to the Tk core that enables mega-widgets to behave indistinguishably from native widgets. This change permits some platforms to implement native look and feel using system widgets, and other platforms to implement the same functionality with mega-widgets. Scripts will be able to run unchanged with either type of widget, with no code changes required.

References

1. M. J. McLennan, "[incr TK]: Building Extensible Widgets with [incr Tcl]," in *1994 Tcl/Tk Workshop Conference Proceedings*, AT&T Bell Laboratories, Allentown Pa., July, 1994.
2. M. J. McLennan, "[incr Tcl]: Object Oriented Programming in Tcl," in *1993 Tcl/Tk Workshop Conference Proceedings*, AT&T Bell Laboratories, Allen-

town Pa., June, 1993.

3. Ioi K. Lam, "Designing Mega Widgets in the Tix Library," in *1995 Tcl/Tk Workshop Conference Proceedings*, Computer Graphics Laboratory, University of Pennsylvania, July, 1995.
4. Mark L. Ulferts, "[incr Widgets] An Object Oriented Mega-Widget Set," in *1995 Tcl/Tk Workshop Conference Proceedings*, DSC Communications Corp., Switching Products Division, July, 1995.
5. Ray Johnson and Scott Stanton, "Cross Platform Support in Tk," in *1995 Tcl/Tk Workshop Conference Proceedings*, Sun Microsystems Laboratories, July, 1995.
6. Shannon Jaeger, "Mega-widgets in Tcl/Tk: Evaluation and Analysis," in *1995 Tcl/Tk Workshop Conference Proceedings*, Department of Computer Science, University of Calgary, July, 1995.

Acknowledgements

Ken Corey created the prototype implementation, and Scott Stanton helped winnow the changes to TK to the smallest number that still do the job.

SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
beazley@cs.utah.edu

Abstract

I present SWIG (Simplified Wrapper and Interface Generator), a program development tool that automatically generates the bindings between C/C++ code and common scripting languages including Tcl, Python, Perl and Guile. SWIG supports most C/C++ datatypes including pointers, structures, and classes. Unlike many other approaches, SWIG uses ANSI C/C++ declarations and requires the user to make virtually no modifications to the underlying C code. In addition, SWIG automatically produces documentation in HTML, LaTeX, or ASCII format. SWIG has been primarily designed for scientists, engineers, and application developers who would like to use scripting languages with their C/C++ programs without worrying about the underlying implementation details of each language or using a complicated software development tool. This paper concentrates on SWIG's use with Tcl/Tk.

1 Introduction

SWIG (Simplified Wrapper and Interface Generator) is a software development tool that I never intended to develop. At the time, I was trying to add a data analysis and visualization capability to a molecular dynamics (MD) code I had helped develop for massively parallel supercomputers at Los Alamos National Laboratory [Beazley, Lomdahl]. I wanted to provide a simple, yet flexible user interface that could be used to glue various code modules together and an extensible scripting language seemed like an ideal solution. Unfortunately there were constraints. First, I didn't want to hack up 4-years of code development trying to fit our MD code into yet another interface scheme (having done so several times already). Secondly, this code was routinely run on systems ranging from Connection

Machines and Crays to workstations and I didn't want to depend on any one interface language—out of fear that it might not be supported on all of these platforms. Finally, the users were constantly adding new code and making modifications. I needed a flexible, yet easy to use system that did not get in the way of the physicists.

SWIG is my solution to this problem. Simply stated, SWIG automatically generates all of the code needed to bind C/C++ functions with scripting languages using only a simple input file containing C function and variable declarations. At first, I supported a scripting language I had developed specifically for use on massively parallel systems. Later I decided to rewrite SWIG in C++ and extend it to support Tcl, Python, Perl, Guile and other languages that interested me. I also added more data-types, support for pointers, C++ classes, documentation generation, and a few other features.

This paper provides a brief overview of SWIG with a particular emphasis on Tcl/Tk. However, the reader should remain aware that SWIG works equally well with Perl and other languages. It is not my intent to provide a tutorial or a user's guide, but rather to show how SWIG can be used to do interesting things such as adding Tcl/Tk interfaces to existing C applications, quickly debugging and prototyping C code, and building interface-language-independent C applications.

2 Tcl and Wrapper Functions

In order to add a new C or C++ function to Tcl, it is necessary to write a special "wrapper" function that parses the function arguments presented as ASCII strings by the Tcl interpreter into a representation that can be used to call the C function. For example,

if you wanted to add the factorial function to Tcl, a wrapper function might look like the following :

```
int wrap_fact(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]) {
    int result;
    int arg0;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    arg0 = atoi(argv[1]);
    result = fact(arg0);
    sprintf(interp->result, "%d", result);
    return TCL_OK;
}
```

In addition to writing the wrapper function, a user will also need to write code to add this function to the Tcl interpreter. In the case of Tcl 7.5, this could be done by writing an initialization function to be called when the extension is loaded dynamically. While writing a wrapper function usually is not too difficult, the process quickly becomes tedious and error prone as the number of functions increases. Therefore, automated approaches for producing wrapper functions are appealing—especially when working with a large number of C functions or with C++ (in which case the wrapper code tends to get more complicated).

3 Prior Work

The idea of automatically generating wrapper code is certainly not new. Some efforts such as Itcl++, Object Tcl, or the XS language included with Perl5, provide a mechanism for generating wrapper code, but require the user to provide detailed specifications, type conversion rules, or use a specialized syntax [Heidrich, Wetherall, Perl5]. Large packages such as the Visualization Toolkit (vtk) may use their own C/C++ translators, but these almost always tend to be somewhat special purpose (in fact, SWIG started out in this manner) [vtk]. If supporting multiple languages is the ultimate goal, a programmer might consider a package such as ILU [Janssen]. Unfortunately, this requires the user to provide specifications in IDL—a process which is unappealing to many users. SWIG is not necessarily intended to compete with these approaches, but rather is designed to be a no-nonsense tool that scientists and engineers can use to easily add Tcl and other scripting languages to their own applications. SWIG is also very different than Embedded Tk (ET) which

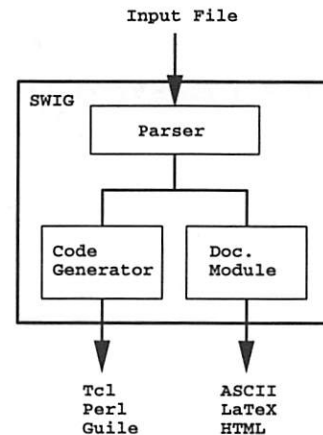


Figure 1: SWIG organization.

also aims to simplify code development [ET]. Unlike ET, SWIG is designed to integrate C functions into Tcl/Tk as opposed to integrating Tcl/Tk into C programs.

4 A Quick Tour of SWIG

4.1 Organization

Figure 1 shows the structure of SWIG. At the core is a YACC parser for reading input files along with some utility functions. To generate code, the parser calls about a dozen functions from a generic language class to do things like write a wrapper function, link a variable, wrap a C++ member function, etc... Each target language is implemented as a C++ class containing the functions that emit the resulting C code. If an “empty” language definition is given to SWIG, it will produce no output. Thus, each language class can be implemented in almost any manner. The documentation system is implemented in a similar manner and can currently produce ASCII, LaTeX, or HTML output. As output, SWIG produces a C file that should be compiled and linked with the rest of the code and a documentation file that can be used for later reference.

4.2 Interface Files

As input, SWIG takes a single input file referred to as an “interface file.” This file contains a few SWIG specific directives, but otherwise contains ANSI C function and variable declarations. Unlike the approach in [Heidrich], no type conversion rules are needed and all declarations are made using familiar ANSI C/C++ prototypes. The following code

shows an interface file for wrapping a few C file I/O and memory management functions.

```
/* File : file.i */
%module fileio
%{
#include <stdio.h>
%}

FILE *fopen(char *filename, char *type);
int fclose(FILE *stream);
typedef unsigned int size_t
size_t fread(void *ptr, size_t size,
             size_t nobj, FILE *stream);
size_t fwrite(void *ptr, size_t size,
             size_t nobj, FILE *stream);
void *malloc(size_t nbytes);
void free(void *);
```

The `%module` directive sets the name of the initialization function. This is optional, but is recommended if building a Tcl 7.5 module. Everything inside the `%{, %}` block is copied directly into the output, allowing the inclusion of header files and additional C code. Afterwards, C/C++ function and variable declarations are listed in any order. Building a new Tcl module is usually as easy as the following :

```
unix > swig -tcl file.i
unix > gcc file_wrap.c -I/usr/local/include
unix > ld -shared file_wrap.o -o Fileio.so
```

4.3 A Tcl Example

Newly added functions work like ordinary Tcl procedures. For example, the following Tcl script copies a file using the binary file I/O functions added in the previous example :

```
proc filecopy {name1 name2} {
    set buffer [malloc 8192];
    set f1 [fopen $name1 r];
    set f2 [fopen $name2 w];
    set nbytes [fread $buffer 1 8192 $f1];
    while {$nbytes > 0} {
        fwrite $buffer 1 $nbytes $f2;
        set nbytes [fread $buffer 1 8192 $f1];
    }
    fclose $f1;
    fclose $f2;
    free $buffer
}
```

4.4 Datatypes and Pointers

SWIG supports the basic datatypes of `int`, `short`, `long`, `float`, `double`, `char`, and `void` as well as

signed and unsigned integers. SWIG also allows derived types such as pointers, structures, and classes, but these are all encoded as pointers. If an unknown type is encountered, SWIG assumes that it is a complex datatype that has been defined earlier. No attempt is made to figure out what data that datatype actually contains or how it should be used. Of course, this is only possible since SWIG's mapping of complex types into pointers allows them to be handled in a uniform manner. As a result, SWIG does not normally need any sort of type-mapping, but `typedef` can be used to map any of the built-in datatypes to new types if desired.

SWIG encodes pointers as hexadecimal strings with type-information. This type information is used to provide a run-time type checking mechanism. Thus, a typical SWIG pointer looks something like the following :

```
_1008e614_Vector_p
```

If this pointer is passed into a function requiring some other kind of pointer, SWIG will generate a Tcl error and return an error message. The NULL pointer is represented by the string "NULL". The SWIG run-time type checker is savvy to typedefs and the relationship between base classes and derived classes in C++. Thus if a user specifies

```
typedef double Real;
```

the type checker knows that `Real *` and `double *` are equivalent (more on C++ in a minute). From the point of view of other Tcl extensions, SWIG pointers should be seen as special "handles" except that they happen to contain the pointer value and its type.

To some, this approach may seem horribly restrictive (or error prone), but keep in mind that SWIG was primarily designed to work with existing C applications. Since most C programs pass complex datatypes around by reference this technique works remarkably well in practice. Run time type-checking also eliminates most common crashes by catching stupid mistakes such as using a wrong variable name or forgetting the "\$" character in a Tcl script. While it is still possible to crash Tcl by forging a SWIG pointer value (or making a call to buggy C code), it is worth emphasizing that existing Tcl extensions may also crash if given an invalid handle.

4.5 Global Variables and Constants

SWIG can install global C variables and constants using Tcl's variable linkage mechanism. Variables

may also be declared as "read only" within the Tcl interpreter. The following example shows how variables and constants can be added to Tcl :

```
// SWIG file with variables and constants
%{
%}

// Some global variables
extern int My_variable;
extern char *default_path;
extern double My_double;

// Some constants
#define PI 3.14159265359
#define PI_4 PI/4.0
enum colors {red,blue,green};
const int SIZEOF_VECTOR = sizeof(Vector);

// A read only variable
%readonly
extern int Status;
%readwrite
```

4.6 C++ Support

The SWIG parser can handle simple C++ class definitions and supports public inheritance, virtual functions, static functions, constructors and destructors. Currently, C++ translation is performed by politely transforming C++ code into C code and generating wrappers for the C functions. For example, consider the following SWIG interface file containing a C++ class definition:

```
%module tree
%{
#include "tree.h"
%}

class Tree {
public:
    Tree();
    ~Tree();
    void insert(char *item);
    int search(char *item);
    int remove(char *item);
    static void print(Tree *t);
};
```

When translated, the class will be access used the following set of functions (created automatically by SWIG).

```
Tree *new_Tree();
void delete_Tree(Tree *this);
void Tree_insert(Tree *this, char *item);
```

```
int Tree_search(Tree *this, char *item);
int Tree_remove(Tree *this, char *item);
void Tree_print(Tree *t);
```

All C++ functions wrapped by SWIG explicitly require the `this` pointer as shown. This approach has the advantage of working for all of the target languages. It also makes it easier to pass objects between other C++ functions since every C++ object is simply represented as a SWIG pointer. SWIG does not support function overloading, but overloaded functions can be resolved by renaming them with the SWIG `%name` directive as follows:

```
class List {
public:
    List();
    %name(ListMax) List(int maxsize);
    ...
}
```

The approach used by SWIG is quite different than that used in systems such as Object Tcl or vtk [vtk, Wetherall]. As a result, users of those systems may find it to be confusing. However, It is important to note that the modular design of SWIG allows the user to completely redefine the output behavior of the system. Thus, while the current C++ implementation is quite different than other systems supporting C++, it would be entirely possible write a new SWIG module that wrapped C++ classes into a representation similar to that used by Object Tcl (in fact, in might even be possible to use SWIG to produce the input files used for Object Tcl).

4.7 Multiple Files and Code Reuse

An essential feature of SWIG is its support for multiple files and modules. A SWIG interface file may include another interface file using the `%include` directive. Thus, an interface for a large system might be broken up into a collection of smaller modules as shown

```
%module package
%{
#include "package.h"
%}

#include geometry.i
#include memory.i
#include network.i
#include graphics.i
#include physics.i

#include wish.i
```

Common operations can be placed into a SWIG library for use in all applications. For example, the `%include wish.i` directive tells SWIG to include code for the `TclAppInit()` function needed to rebuild the `wish` program. The library can also be used to build modules allowing SWIG to be used with common Tcl extensions such as Expect [Expect]. Of course, the primary use of the library is with large applications such as Open-Inventor which contain hundreds of modules and a substantial class hierarchy [Invent]. In this case a user could use SWIG's include mechanism to selectively pick which modules they wanted to use for a particular problem.

4.8 The Documentation System

SWIG produces documentation in ASCII, LaTeX, or HTML format describing everything that was wrapped. The documentation follows the syntax rules of the target language and can be further enhanced by adding descriptions in a C/C++ comment immediately following a declaration. These comments may also contain embedded LaTeX or HTML commands. For example:

```
extern size_t fread(void *ptr, size_t size,
                    size_t nobj, FILE *stream);
/* {\tt fread} reads from {\tt stream} into
the array {\tt ptr} at most {\tt nobj} objects
of size {\tt size}. {\tt fread} returns
the number of objects read. */
```

When output by SWIG and processed by LaTeX, this appears as follows :

```
size_t : fread ptr size nobj stream

fread reads from stream into the array ptr at
most nobj objects of size size. fread returns
the number of objects read.
```

4.9 Extending the SWIG System

Finally, SWIG itself can be extended by the user to provide new functionality. This is done by modifying an existing or creating a new language class. A typical class must specify the following functions that determine the behavior of the parser output :

```
// File : swigtcl.h
class TCL : public Language {
private:
    // Put private stuff here
public :
    TCL();
```

```
int main(int, char *argv[]);
void create_function(char *,char *,DataType*,
                    ParmList *);
void link_variable(char *,char *,DataType *);
void declare_const(char *,int,char *);
void initialize(void);
void headers(void);
void close(void);
void usage_var(char *,DataType*,char **);
void usage_func(char *,DataType*,ParmList*,
                char **);
void usage_const(char *,int,char*,char**);
void set_module(char *);
void set_init(char *);
};
```

Descriptions of these functions can be found in the SWIG users manual. To build a new version of SWIG, the user only needs to provide the function definitions and a main program which looks something like the following :

```
// SWIG main program

#include "swig.h"
#include "swigtcl.h"

int main(int argc, char **argv) {

    Language *lang;

    lang = new TCL;
    SWIG_main(argc,argv,lang,(Documentation *) 0);

}
```

When linked with a library file, any extensions and modifications can now be used with the SWIG parser. While writing a new language definition is not entirely trivial, it can usually be done by just copying one of the existing modules and modifying it appropriately.

5 Examples

5.1 A C-Tcl Linked List

SWIG can be used to build simple data structures that are usable in both C and Tcl. The following code shows a SWIG interface file for building a simple linked list.

```
/* File : list.i */
%{
struct Node {
    Node(char *n) {
```



```

    name = new char[strlen(n)+1];
    strcpy(name,n);
    next = 0;
};
char *name;
Node *next;
};
%}

```

```

// Just add struct definition to
// the interface file.

```

```

struct Node {
    Node(char *);
    char *name;
    Node *next;
};

```

When used in a Tcl script, we can now create new nodes and access individual members of the Node structure. In fact, we can write code to convert between Tcl lists and linked lists entirely in Tcl as shown :

```

# Builds linked list from a Tcl list
proc buildlist {list head} {
    set nitems [llength $list];
    for {set i 0} {$i < $nitems} {incr i -1} {
        set item [lrange $list $i $i]
        set n [new_Node $item]
        Node_set_next $n $head
        set head $n
    }
    return $head
}
# Builds a Tcl list from a linked list
proc get_list {l1list} {
    set list {}
    while {$l1list != "NULL"} {
        lappend list [Node_name_get $l1list]
        set l1list [Node_get_next $l1list]
    }
    return $list
}

```

When run interactively, we could now use our Tcl functions as follows.

```

% set l {John Anne Mary Jim}
John Anne Mary Jim
% set l1 [buildlist $l _0_Node_p]
_1000cab8_Node_p
% get_list $l1
Jim Mary Anne John
% set l1 [buildlist {Mike Peter Dave} $l1]
_1000cc38_Node_p
% get_list $l1
Dave Peter Mike Jim Mary Anne John
%

```

Aside from the pointer values, our script acts like any other Tcl script. However, we have built up a real C data structure that could be easily passed to other C functions if needed.

5.2 Using C Data-Structures with Tk

In manner similar to the linked list example, Tcl/Tk can be used to build complex C/C++ data structures. For example, suppose we wanted to interactively build a graph of "Nodes" for use in a C application. A typical interface file might include the following functions:

```

%{
#include "nodes.h"
%}

%include wish
extern Node *new_node();
extern void AddEdge(Node *n1, Node *n2);

```

Within a Tcl/Tk script, loosely based on one to make ball and stick graphs in [Ousterhout], a graph could be built as follows:

```

proc makeNode {x y} {
    global nodeX nodeY nodeP edgeFirst edgeSecond
    set new [.create oval [expr $x-15] \
        [expr $y-15] [expr $x+15] \
        [expr $y+15] -outline black \
        -fill white -tags node]
    set newnode [new_node]
    set nodeX($new) $x
    set nodeY($new) $y
    set nodeP($new) $newnode
    set edgeFirst($new) {}
    set edgeSecond($new) {}
}

proc makeEdge {first second} {
    global nodeX nodeY nodeP edgeFirst edgeSecond
    set x1 $nodeX($first); set y1 $nodeY($first)
    set x2 $nodeX($second); set y2 $nodeY($second)
    set edge [.c create line $x1 $y1 $x2 $y2 \
        -tags edge]
    .c lower edge
    lappend edgeFirst($first) $edge
    lappend edgeSecond($first) $edge
    AddEdge $nodeP($first) $nodeP($second)
}

```

These functions create Tk canvas items, but also attach a pointer to a C data structure to each one. This is done by maintaining an associative array mapping item identifiers to pointers (with the nodeP() array). When a particular "node" is referenced later, we can use this to get its pointer use it in calls to C functions.

5.3 Parsing a C++ Simple Class Hierarchy

As mentioned earlier, SWIG can handle C++ classes and public inheritance. The following example provides a few classes and illustrates how this is accomplished (some code has been omitted for readability).

```
// A SWIG inheritance example
%module shapes
%{
#include "shapes.h"
%}

class Shape {
private:
    double xc, yc;
public:
    virtual double area() = 0;
    virtual double perimeter() = 0;
    void    set_position(double x, double y);
    void    print_position();
};

class Circle: public Shape {
private:
    double radius;
public:
    Circle(double r);
    double area();
    double perimeter();
};

class Square : public Shape {
private:
    double width;
public:
    Square(double w);
    double area();
    double perimeter();
};
```

Now, when wrapped by SWIG ¹, we can use our class structure as follows:

```
% set c [new_Circle 4]
_1000ad70_Circle_p
% set s [new_Square 10]
_1000adc0_Square_p
% Shape_area $c
50.26548246400000200
% Shape_area $s
100.000000000000000000
```

¹When parsing C++ classes, SWIG throws away everything declared as private, inline code, and a lot of the other clutter found in C++ header files. Primarily this is provided only to make it easier to build interfaces from existing C++ header files.

```
% Shape_set_position $c -5 10
% Circle_print_position $c
xc = -5, yc = 10
%
```

In our example, we have created new **Circle** and **Square** objects, but these can be used interchangeably in any functions defined in the **Shape** base class. The SWIG type checker is encoded with the class hierarchy and knows the relationship between the different classes. Thus, while an object of type **Circle** is perfectly acceptable to a function operating on shapes, it would be unacceptable to a function operating only on the **Square** type. As in C++, any functions in the base class can be called in the derived class as shown by the **Circle.print_position** function above.

6 Using SWIG in Real Applications

So far only a few simple toy examples have been presented to illustrate the operation of SWIG in general. This section will describe how SWIG can be used with larger applications.

6.1 Use in Scientific Applications

Many users, especially within the scientific and engineering community, have spent years developing simulation codes. While many of these users appreciate the power that a scripting language can provide, they don't want to completely rewrite their applications or spend all of their time trying to build a user-interface (most users would rather be working on the scientific problem at hand). While SWIG certainly won't do everything, it can dramatically simplify this process.

As an example, the first SWIG application was the SPaSM molecular dynamics code developed at Los Alamos National Laboratory [Beazley, Lomdahl]. This code is currently being used for materials science problems and consists of more than 200 C functions and about 20000 lines of code. In order to use SWIG, only the **main()** function had to be rewritten along with a few other minor modifications. The full user interface is built using a collection of modules for simulation, graphics, memory management, etc... A user may also supply their own interface modules—allowing the code to be easily extended with new functionality capabilities as needed. All of the interface files, containing a few hundred lines of function declarations, are automatically translated

into more than 2000 lines of wrapper code at compile time. As a result, many users of the system know how to extend it, but are unaware of the actual wrapping procedure.

After modifying the SPaSM code to use SWIG, most of the C code remained unchanged. In fact, in subsequent work, we were able to eliminate more than 25% of the source code—almost all of which was related to the now obsolete interface method that we replaced. More importantly, we were able to turn a code that was difficult to use and modify into a flexible package that was easy to use and extend. This has had a huge impact, which can not be understated, on the use of the SPaSM code to solve real problems.

6.2 Open-GL and Inventor

While SWIG was primarily designed to work with application codes, it can also be used to wrap large libraries. At the University of Utah, Peter-Pike Sloan used SWIG to wrap the entire contents of the Open-GL library into Tcl for use with an Open-GL Tk widget. The process of wrapping Open-GL with SWIG was as simple as the following:

- Make a copy of the `gl.h` header file.
- Clean it up by taking a few C preprocessor directives out of it. Fix a few typedefs.
- Insert the following code into the beginning


```
%module opengl
%{
#include <GL/gl.h>
%}
```

... Copy edited `gl.h` here ...
- Modify the Open-GL widget to call `Opengl_Init`.
- Recompile

The entire process required only about 10 minutes of work, but resulted in more than 500 constants and 360 functions being added to Tcl. Furthermore, this extension allows Open-GL commands to be issued directly from Tcl's interpreted environment as shown in this example (from the Open-GL manual [OpenGL]).

```
% ... open GL widget here ...
% glClearColor 0.0 0.0 0.0 0.0
```

```
% glClear $GL_COLOR_BUFFER_BIT
% glColor3f 1.0 1.0 1.0
% glOrtho -1.0 1.0 -1.0 1.0 -1.0 1.0
% glBegin $GL_POLYGON
%   glVertex2f -0.5 -0.5
%   glVertex2f -0.5 0.5
%   glVertex2f 0.5 0.5
%   glVertex2f 0.5 -0.5
% glEnd
% glFlush
```

Early work has also been performed on using SWIG to wrap portions of the Open-Inventor package [Invent]. This is a more ambitious project since Open-Inventor consists of a very large collection of header files and C++ classes. However, the SWIG library mechanism can be used effectively in this case. For each Inventor header, we can create a sanitized SWIG interface file (removing a lot of the clutter found in the header files). These interface files can then be organized to mirror the structure of the Inventor system. To build a module for Tcl, a user could simply specify which modules they wanted to use as follows:

```
%module invent
%{
... put headers here ...
%}

#include "Inventor/Xt/SoXt.i"
#include "Inventor/Xt/SoXtRenderArea.i"
#include "Inventor/nodes/SoCone.i"
#include "Inventor/nodes/SoDirectionalLight.i"
#include "Inventor/nodes/SoMaterial.i"
#include "Inventor/nodes/SoPerspectiveCamera.i"
#include "Inventor/nodes/SoSeparator.i"
```

While wrapping the Inventor library will require significantly more work than Open-GL, SWIG can be used effectively with such systems.

6.3 Building Interesting Applications by Cut and Paste

SWIG can be used to construct tools out of dissimilar packages and libraries. In contrast to combining packages at an input level as one might do with Expect, SWIG can be used to build applications at a function level [Expect]². For example, using a simple interface file, you can wrap the C API of MATLAB 4.2 [MATLAB]. This in turn lets you build a Tcl/Tk interface for controlling MATLAB programs. Separately, you could wrap a numerical

²SWIG can also be used to add extensions directly to expect and expecttk.

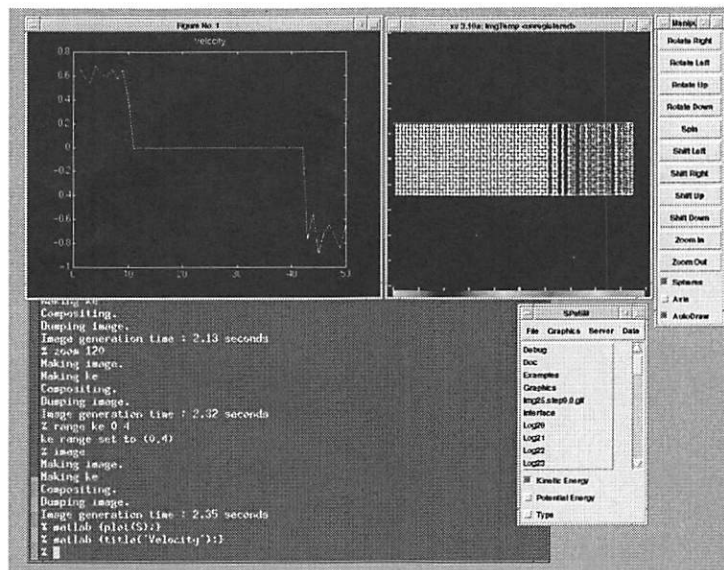


Figure 2: Simple Tcl/Tk Steering Application Built with SWIG

simulation code. Now, a few translation functions could be written and both packages combined into a single Tcl/Tk application. You have now built a simple “computational steering” system that allows you to run a simulation code, visualize data in MATLAB, and control the entire system with a Tcl/Tk based interface. Figure 2 shows a screen snapshot from such a simulation in which the SPaSM molecular dynamics code, a visualization module library, image server (using xv), and MATLAB have been combined. Under this system, the user can interactively set up and run simulations while watching the results evolve in real-time.

6.4 Language Independent Applications

Each scripting language has its own strengths and weaknesses. Rather than trying to choose the language that is the best at everything (which is probably impossible), SWIG allows a user to use the best language for the job at hand. I’m always finding myself writing little utilities and programs to support my scientific computing work. Why should I be forced to use only one language for everything? With SWIG, I can put a Perl interface on a tool and switch over to Tcl/Tk at anytime by just changing a few Makefile options.

Since SWIG provides a language-independent inter-

face specification, it is relatively easy to use SWIG generated modules in a variety of interesting applications. For example, the identical MATLAB module used in the last Tcl example could be imported as Perl5 module and combined with a Perl script to produce graphical displays from Web-server logs as shown in Figure 3.

Some have promoted the idea of encapsulating several languages into a higher level language as has been proposed with Guile [Lord]. This may be fine if one wants to write code that uses different languages all at once, but when I want to use only Tcl or Perl for an application, I almost always prefer using the real versions instead of an encapsulated variant. SWIG makes this possible without much effort.

6.5 Using Tcl as a debugger

Since SWIG can work with existing C code, it can be used quite effectively as a debugger. You can wrap C functions and interact with them from `tclsh` or `wish`. Unlike most traditional debuggers, you can create objects, buffers, arrays, and other things on the fly by putting appropriate definitions in the interface file. Since `tclsh` or `wish` provides a `main()` function, you can even rip small pieces out of a larger package without including that package’s main program. Scripts can be written to test

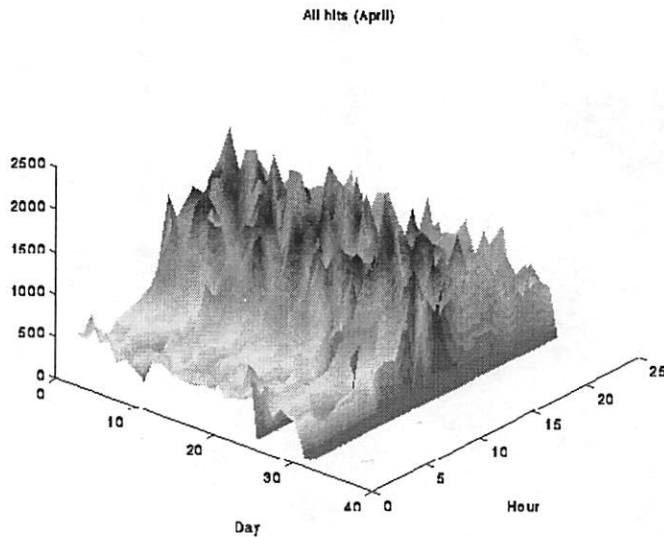


Figure 3: Web-server activity graph. Generated from a Perl script, but uses an unmodified SWIG generated module for integrating MATLAB with Tcl/Tk.

out different parts of your code as it is developed. When you are satisfied with the module, removing the interface is as easy as discarding the SWIG interface file. I have used SWIG as a debugger for developing graphics libraries, network servers, simulation codes, and a wide range of other projects—some of which didn't even use a scripting language when completed.

7 Limitations

SWIG represents a balance of flexibility and ease of use. I have always felt that the tool shouldn't be more complicated than the original problem. Therefore, SWIG certainly won't do everything. While I believe the pointer representation of complex datatypes works well in practice, some users have found this to be too general or confusing. SWIG's support for C++ is also limited by the fact that SWIG does not support operator overloading, multiple inheritance, templates, and a number of other more advanced C++ features. SWIG also lacks support for default or variable length function arguments, array notation, pointers to functions (unless hidden by a typedef) and an exception mechanism. Finally, SWIG does not support all of the features of Tcl such as associative arrays. These

aren't an inherent part of the C language so it is difficult to generalize how they should be handled or generated. I prefer to keep the tool simple while leaving these issues up to the individual programmer.

Fortunately, it is almost always possible to work around many problems by writing special library functions or making modifications to the SWIG language modules to produce the desired effect. For example, when wrapping Open-GI, SWIG installed all of the Open-GI constants as Tcl global variables. Unfortunately, none of these variables were accessible inside Tcl procedures unless they were explicitly declared global. By making a modification to the Tcl language module, it was possible to put the GL constants into hash table and perform a hidden "lookup" inside all of the GL-related functions. Similarly, much of the functionality of SWIG can be placed into library files for use in other modules.

8 Conclusions

SWIG has been in use for approximately one year. At Los Alamos National Laboratory, its use with the SPaSM code has proven to be a remarkably simple, stable, and bug-free way to build and modify user

interfaces. At the University of Utah, SWIG is being used in a variety of other applications where the users have quickly come to appreciate its ease of use and flexibility.

While SWIG may be inappropriate for certain applications, I feel that it opens up Tcl/Tk, Python, Perl, Guile, and other languages to users who would like to develop interesting user interfaces for their codes, but who don't want to worry about the low-level details or figuring out how to use a complicated tool.

Future directions for SWIG include support for other scripting languages, and a library of extensions. SWIG may also be extended to support packages such as incremental Tcl. Work is also underway to port SWIG to non-unix platforms.

9 Acknowledgments

This project would not have been possible without the support of a number of people. Peter Lomdahl, Shujia Zhou, Brad Holian, Tim Germann, and Niels Jensen at Los Alamos National Laboratory were the first users and were instrumental in testing out the first designs. Patrick Tullmann at the University of Utah suggested the idea of automatic documentation generation. John Schmidt, Kurtis Bleeker, Peter-Pike Sloan, and Steve Parker at the University of Utah tested out some of the newer versions and provided valuable feedback. John Buckman suggested many interesting improvements and has been instrumental in the recent development of SWIG. Finally I'd like to thank Chris Johnson and the members of the Scientific Computing and Imaging group at the University of Utah for their continued support and for putting up with my wrapping every software package I could get my hands on. SWIG was developed in part under the auspices of the US Department of Energy, the National Science Foundation, and National Institutes of Health.

10 Availability

SWIG is free software and available via anonymous FTP at

[ftp.cs.utah.edu/pub/beazley/SWIG](ftp://ftp.cs.utah.edu/pub/beazley/SWIG)

More information is also available on the SWIG homepage at

<http://www.cs.utah.edu/beazley/SWIG>

References

- [Beazley] D. M. Beazley and P. S. Lomdahl, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5*, Parallel Comp. 20 (1994) p. 173-195.
- [ET] Embedded Tk,
<ftp://ftp.vnet.net/pub/users/drh/ET.html>
- [Expect] Don Libes, *Exploring Expect*, O'Reilly & Associates, Inc. (1995).
- [Heidrich] Wolfgang Heidrich and Philipp Slusallek, *Automatic Generation of Tcl Bindings for C and C++ Libraries.*, USENIX 3rd Annual Tcl/Tk Workshop (1995).
- [Janssen] Bill Janssen, Mike Spreitzer, *ILU : Inter-Language Unification via Object Modules*, OOPSLA 94 Workshop on Multi-Language Object Models.
- [Lomdahl] P. S. Lomdahl, P. Tamayo, N. Grønbech-Jensen, and D. M. Beazley, *Proc. of Supercomputing 93*, IEEE Computer Society (1993), p. 520-527.
- [Lord] Thomas Lord, *An Anatomy of Guile, The Interface to Tcl/Tk*, Proceedings of the USENIX 3rd Annual Tcl/Tk Workshop (1995).
- [MATLAB] *MATLAB External Interface Guide*, The Math Works, Inc. (1993).
- [Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).
- [Perl5] Perl5 Programmers reference,
<http://www.metronet.com/perlinfo/doc>, (1996).
- [vtk] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*, Prentice Hall (1995).
- [Invent] J. Wernecke, "The Inventor Mentor", Addison-Wesley Publishing (1994).
- [Wetherall] D. Wetherall, C. J. Lindblad, "Extending Tcl for Dynamic Object-Oriented Programming", Proceedings of the USENIX 3rd Annual Tcl/Tk Workshop (1995).
- [OpenGL] J. Neider, T. Davis, M. Woo, "OpenGL Programming Guide", Addison-Wesley Publishing (1993).

Automated Wrapping of a C++ Class Library into Tcl

Ken Martin

General Electric Corporate Research and Development

1 Research Circle, Niskayuna, NY 12309

martink@crd.ge.com

Abstract

In this paper we describe an approach to wrapping a preexisting C++ class library into the interpreted Tcl environment. Specifically, we look at our efforts over the past two years to add the Tcl interpreted environment on top of the Visualization Toolkit which consists of over three hundred C++ classes. We address how we overcame the fundamental issues involved in wrapping existing C++ code and what limitations we had to accept. We contrast our approach to other Tcl object oriented extensions such as Object Tcl and [incr Tcl] and explain why they were not suitable. We conclude by looking at future directions for our work.

1. Introduction

Over ten years ago researchers at General Electric started work on an interpreted, object-oriented visualization system [2]. At the time, C++ was still in its infancy and Smalltalk was considered one of the most promising languages. Taking some of the best features of Smalltalk, a new language was developed called LYMB. This provided an interpreter and script based language very similar to Tcl with the exception that it was built on object-oriented principles. While the system was very valuable, it had some weaknesses that made it monolithic and difficult to integrate into other systems. In late 1993 an effort was started to create a new system that would overcome many of the problems associated with its predecessor.

One thing we had learned was that object-oriented technology was a big productivity enhancer for our group. We decided that any replacement system should be object oriented. We also wanted the system to be portable, extensible and easily integrated into a wide variety of applications. To this end we selected C++ as our development language. About eighteen months into the development of The Visualization Toolkit [5] (VTK) as it later was named, we realized that using only C++ was proving to be a problem. In LYMB we had an inter-

preted environment that made interactive development very quick and easy. With VTK we had no interpreter or scripting language and its loss was profoundly felt. At this point we started looking into adding an interpreter into, or on top of our C++ class library.

Unlike LYMB which used its own interpreted language, we wanted to select a language that was already established. We would focus on data visualization which was our strength, and let others (such as the Tcl community) focus on language issues. The decision came down to Tcl or Python. Tcl's wide acceptance, maturity and easy extensibility won out in spite of the fact that it didn't have any direct object-oriented support.

After some quick research into the Tcl community, we found that there wasn't a fully automatic way to wrap C++ into Tcl without significantly modifying the C++ code. Since we already had over 200 classes written, a serious change in coding style was out of the question. At this point we decided to try writing our own automated C++ code wrapper for Tcl, the result of which is discussed in the remainder of this paper.

2. Related Work

When we started writing our C++ to Tcl wrapper generator there were no suitable alternatives. Now, over two years later, much progress has been made on this front and it is worth revisiting. [incr Tcl][3][4] has emerged as a popular object oriented extension to Tcl providing objects, mega-widgets namespaces and more. While this system does provide strong encapsulation and integration with Tcl, it does not provide a strong interface to C++ classes. [incr Tcl] allows you to bind C functions or static C++ functions into [incr Tcl] classes. It does not support wrapping of C++ classes, construction, destruction of C++ objects, or invocation of C++ member functions. Object Tcl[7] is similar to [incr Tcl] in that it is designed to support object oriented programming for Tcl, not the incorporation of existing C++ libraries.

The Simple Wrapper Interface Generator (SWIG) by Dave Beazley, is a tool for that can generate wrapper code for a number of different target languages including Tcl. SWIG requires an interface file that prototypes the functions that are to be wrapped. For many C libraries the ANSI C prototypes can be directly included into the interface file. For C++ code it isn't as easy. SWIG provides a good wrapper for prototype functions, but it doesn't provide much support for wrapping C++ classes. Creating, destroying, type-conversion and passing of classes as arguments is not supported in the current version. So while SWIG is an excellent package for wrapping C libraries, it currently isn't suitable for wrapping C++ class libraries.

Objective-Tcl[1] does provide much of the functionality we required for wrapping VTK. Since we had decided to use C++ as our compiled language and Objective-Tcl binds to Objective-C, we were not able to use it. In some ways Objective-Tcl is a better solution since it supports creating classes and methods within the Tcl domain. Our solution provides access to the C++ objects, but it does not provide any method for object oriented programming in the Tcl domain. The use of Objective-C's run time class information solves some of the tricky problems of C++ object integration that we had to address.

Otcl[6] is the system that most resembles our solution. In many ways Otcl, like Objective-Tcl, provides a better solution to wrapping existing classes into Tcl. But there are some key differences between Otcl and our approach. Otcl requires the user to specify a Class Description Language (CDL) file for all the objects you want integrated into Tcl. In this file, you must specify the public methods, their arguments and whether they require static or dynamic binding. In our approach the required information is extracted from the C++ header files.

Otcl currently doesn't handle type conversions between superclasses and subclasses which can lead to problems, especially in libraries that make use of multiple inheritance. Early in our development we ran into this problem which prompted our efforts to perform true type conversion for C++ objects. To support VTK's hardware independence, instances are created on the C++ side which must be returned and used on the Tcl side. Otcl supports this through the use of special classes that the C++ side can instantiate. But this means that the original C++ code must be modified to create instances of these special classes instead of its normal behavior. This limitation isn't present in our approach. Otcl also lacks support for overloaded methods which we support.

While Otcl may not provide as tight a wrapping of C++ classes, it does provide much more flexibility in the Tcl domain. C++ classes can be subclassed in the Tcl domain, passed back and forth, and methods added or overridden. Our work does not support any object oriented extensions or subclassing from the Tcl domain.

3. Methods

There are a number of difficult issues to be resolved in automatically wrapping C++ into Tcl. We will examine them in the order that we dealt with them. The first problem that we ran into was parsing our C++ code. In order to wrap it into Tcl, we needed a method of parsing the code and breaking it into its lexical components. To do this we decided to use the traditional tools LEX and YACC.

As it turns out, writing LEX and YACC code for C++ is not an easy task since C++ has so many features. There were two ideas that enabled us to get by. The first was our philosophy regarding C++. We believed in using only the core features of C++ within a relatively rigid development environment. We did not use templates, run time type checking or exceptions. Our coding standards dictated that all class names start with a common prefix (vtk) a limit of one class per file, and a set of simple macros for performing common set/get methods. With these restrictions we were able to develop a LEX and YACC based program that could parse our header files and extract the pertinent information. We did not make an attempt to handle preprocessor directives, which means that our wrapping of a class is done based on that class's header file alone, without looking at its superclasses' header files.

The first step in wrapping a class library is to determine which classes are concrete and which are abstract. In object-oriented terminology, abstract classes define an API for their subclasses. They are not meant to be instantiated and under some circumstances it can be a compiler error. To resolve this we modified our LEX and YACC code to parse the header files and look for any pure virtual functions, which indicate that a class is abstract. We then create a list of abstract classes and concrete classes. This approach is not fool proof. There may be a class whose superclass is abstract, that hasn't made itself concrete by providing a concrete implementation of the pure virtual function of the superclass. Since we judge each class based solely on its header file, we cannot detect this condition. In this case, which has happened only twice in our now 360 classes, we must manually specify that the class is abstract.

```

int vtkNewInstanceCommand(ClientData cd, Tcl_Interp *interp,
                          int argc, char *argv[])
{
    Tcl_HashEntry *entry;
    int is_new;
    char temps[80];

    if (argc != 2)
    {
        interp->result = "vtk object creation requires one argument, a name.";
        return TCL_ERROR;
    }

    if ((argv[1][0] >= '0') && (argv[1][0] <= '9'))
    {
        interp->result = "vtk instances names must start with a letter.";
        return TCL_ERROR;
    }

    if (Tcl_FindHashEntry(&vtkInstanceLookup, argv[1]))
    {
        interp->result = "a vtk instance with that name already exists.";
        return TCL_ERROR;
    }

    /* Create an instance of vtkActor */
    if (!strcmp("vtkActor", argv[0]))
    {
        ClientData temp = vtkActorNewCommand();
        entry = Tcl_CreateHashEntry(&vtkInstanceLookup, argv[1], &is_new);
        Tcl_SetHashValue(entry, temp);
        sprintf(temps, "%x", (void *)temp);
        entry = Tcl_CreateHashEntry(&vtkPointerLookup, temps, &is_new);
        Tcl_SetHashValue(entry, (ClientData)(strdup(argv[1])));
        Tcl_CreateCommand(interp, argv[1], vtkActorCommand, temp,
                          (Tcl_CmdDeleteProc *)vtkTclGenericDeleteObject);
        entry = Tcl_CreateHashEntry(&vtkCommandLookup, argv[1], &is_new);
        Tcl_SetHashValue(entry, (ClientData)(vtkActorCommand));
    }

    /* Create an instance of vtkAppendPolyData */
    if (!strcmp("vtkAppendPolyData", argv[0]))
    {
        ...
    }
    ...
}

```

Listing 1. C++ instance creation code for Tcl.

For all of the concrete classes we run a second LEX/YACC program to generate the required Tcl code to create a new instance. We use the same model as the Tk widgets in that the class name serves as the command to create an instance of that class. In our VTK_Init function, we simply create a command using Tcl_CreateCommand for every concrete class in the library. All of these commands invoke the same function vtkNewInstanceCommand, which checks to make sure that the instance name was specified and is unique. It then creates a new command with the same name as the instance, and attaches it to a function that is responsible for handling methods on that class. It also creates a C++ instance of the desired class and associates the Tcl instance name with the Tcl method function for that class and the C++ object pointer. We use three Tcl hash tables to store these associations.

The code sample in Listing 1, shows a small portion of the vtkNewInstanceCommand function. In this example it shows the logic for creating an instance of the class vtkActor. First it does some quick checks to make sure that the desired instance name is specified, starts with a letter, and isn't already in use. Then it does a string compare for argv[0] against "vtkActor". If it matches, it calls a function called vtkActorNewCommand which will create an instance of vtkActor and then return it as type ClientData. The next two lines create an association between the Tcl name for that instance and the actual C++ pointer address. This is stored in the vtkInstanceLookup hash table. The next three lines perform a similar task, associating the pointer address with the Tcl instance name. The next line creates a Tcl command with the same name as this instance. It also associates the vtkActorCommand function with this instance to handle any method invocations. The final two lines create an association between the Tcl instance name and the function that should be used for invoking methods.

The next step in the process creates the method functions for all of the classes. Where we only allow the user to create instances of concrete classes, we must provide support for invoking methods from both concrete classes or their abstract superclasses. This is because some of the methods of a concrete class may be implemented in an abstract superclass. So we must wrap methods of the abstract classes even though we do not allow the user to create a direct instance of one. The method function serves to connect the Tcl commands with their string arguments, to the C++ method invocations. There are four main operations that occur in these functions: typecasting which we will discuss later, chaining up the class hierarchy to search for unresolved methods, additional methods that are specific to the Tcl

interpreted environment, and the bulk of the code serves to handle the invocation of the C++ methods.

Chaining up the hierarchy is handled in a simple manner. If a method wasn't found in the current function, we then invoke the same Tcl command, but this time using the superclass's method function. This continues until either the method is found or the top of the hierarchy is reached. For classes with multiple inheritance we perform a depth first search of the inheritance tree. If we cannot find a method that matches then we return TCL_ERROR. Since Tcl is an interactive language, we use the method function to add two additional commands to each class. We provide a hook for the Tcl user to invoke the PrintSelf method on a class and have it be returned in interp->result. We also added a command to list all the possible methods you can send to a class and its superclasses.

As we mentioned above, the bulk of the method function handles wrapping the C++ methods. From the C++ header file we obtain the names of the methods and the number and type of arguments that they take and return. For each method we first compare the Tcl method name, stored as argv[1], with the target method. We then check to see if the number of arguments matches. After this we start taking apart the Tcl string arguments and converting them into the appropriate C++ arguments. For the traditional C data types we use the standard Tcl functions such as Tcl_GetDouble. For passing C++ objects we use our own function. If any of the argument conversions fail, we assume that this wasn't the correct method and we continue searching for a match. This is how we handle overloaded functions. Two functions may have the same name and the same argument count, but the string arguments passed in from Tcl may convert correctly for one method and not the other, effectively disambiguating them. The return value of a method is converted into a string and then returned in interp->result. The example in Listing 2, from vtkActor's method function will help clarify this process.

The first line checks to make sure the method name matches and that the argument count is correct. Remember that the first Tcl argument is the instance name, the second is the method name and the third Tcl argument is the first C++ argument. So if we need three C++ arguments then we check to see if argc is five. The next series of statements try to convert the Tcl arguments into the desired type for C++, in this case floats. If all the arguments convert correctly, we invoke the function and return TCL_OK. In this example the C++ method does not have a return value. The C++ instance, stored in the variable op, is passed in as the ClientData from Tcl.

```

int vtkActorCxxCommand(vtkActor *op, Tcl_Interp *interp,
                       int argc, char *argv[])
{
    int    tempi;
    double tempd;
    static char temps[80];
    int    error;

    // some type conversion routines etc
    ...

    // check for an invocation of the SetPosition method
    if ((!strcmp("SetPosition",argv[1]))&&(argc == 5))
    {
        float    temp0;
        float    temp1;
        float    temp2;
        error = 0;

        if (Tcl_GetDouble(interp,argv[2],&tempd) == TCL_OK)
            { temp0 = tempd; }
        else
            { error = 1; }
        if (Tcl_GetDouble(interp,argv[3],&tempd) == TCL_OK)
            { temp1 = tempd; }
        else
            { error = 1; }
        if (Tcl_GetDouble(interp,argv[4],&tempd) == TCL_OK)
            { temp2 = tempd; }
        else
            { error = 1; }
        if (!error)
        {
            op->SetPosition(temp0,temp1,temp2);
            interp->result[0] = '\0';
            return TCL_OK;
        }
    }

    // check other methods
    ...

    // If we haven't found a match, try chain up the superclasses
    if (vtkObjectCxxCommand((vtkObject *)op,interp,argc,argv) == TCL_OK)
    {
        return TCL_OK;
    }

    ...
}

```

Listing 2. Excerpt from vtkActor's method function.

The first concern with this approach is that C++ header files do not provide enough information to correctly convert to and from Tcl strings. The classic example of this is a method that returns a pointer to a float. The C++ header file provides no information about how many elements are in that array. In some cases the correct size may not be known until runtime, but for many cases the return size is fixed and should be used in the wrapping process.

To support this we create a hint file that contains a series of lines of the form: class name, method name, return type and expected size. When the wrapping code encounters a method that it normally couldn't wrap, it checks the hint file to see if the required information is in there. As it turns out, our hint file is only one or two pages long. This is because many of our methods that return fixed size arrays are done with macros which include the size of the result in the macro call. These macros are typically Set/Get methods for instance variables such as the (X, Y, Z) position of an actor. These same macros allow us to pass arrays into C++ methods since the required size of the array is known at the time of wrapping.

Passing C++ objects to and from methods is where a great portion of the complexity comes from. A string passed to a method that is expecting an object can be looked up in the hash tables defined earlier. If it is found, we then have to perform a proper C++ typecasting to the desired argument type. For example, say that you have a class B that is a subclass of class A. You have created an instance of B named foo from the Tcl interpreter, and you want to pass it into a method that takes an argument of type A. To do this we first look up the instance foo in our hash tables to get the C++ object pointer. Then we invoke the method function for that class (B in this example) asking it to perform a typecasting to a result of type A. We obtain the method function by looking it up in a hash table. If the typecasting is possible, e.g. if B is really a subclass of A, then the correct pointer is returned and the method will be invoked. Otherwise an error occurs and method resolution continues as with any other argument mismatch.

An example illustrates why this typecasting is so important. Given the following piece of C++ code where B is a subclass of A, you might expect that bar and foo would point to the same location in memory, but in many cases they will not. Doing a blind typecasting of a C++ object obtained from the hashtable into the desired argument type will create all sorts of problems, especially since no checking will be done to ensure that the two C++ types are even compatible.

```
class A ...;
class B: A ...;
B *foo;
A *bar = foo;
```

It also happens that some C++ methods create instances of C++ classes and return them. In this case, there will not be a Tcl string name for the instance since it was created on the C++ side. When a C++ method returns a C++ class pointer, we look up that pointer in the hash tables. If it has an associated name, then we return that string. If it doesn't then we generate a name of the form vtkTemp1 or vtkTemp2 etc. We then enter that name into the hash tables as if the user had created the instance. This is commonly used in situations like the following Tcl script:

```
set activeCamera [aRenderer GetActiveCamera];
```

We set the value of the variable activeCamera to the generated name returned from the GetActiveCamera method. When it comes to freeing instances, we use the rule that if the instance was created from the Tcl side, then we will free the C++ side when the Tcl side is freed. If the instance was created on the C++ side, then we require that whatever class created the instance, also free it.

Now that we have discussed how we pass primitive data types, arrays and C++ objects, there is one more type of argument that is worth mentioning: the user defined function. User defined functions are essentially callbacks that certain classes support. The user specifies a function to be called and an argument to pass to that function whenever a certain event occurs. With a Tcl wrapping we don't want to pass a function pointer, but rather a string of Tcl commands to be executed. So we package up the string and the interpreter that sent the command into a structure, and set that as the argument to be sent to the function. We then specify that the user defined function call the generic function below. This function takes apart the structure and invokes Tcl_GlobalEval to execute the string of Tcl script.

```
void vtkTclVoidFunc(void *arg)
{
    vtkTclVoidFuncArg *arg2;
    arg2 = (vtkTclVoidFuncArg *)arg;
    Tcl_GlobalEval(arg2->interp,
                  arg2->command);
}
```

From Tcl, a user defined function can be setup in the following straightforward manner.

```

# tcl code to draw a cube
# create a few instances of vtk classes
vtkRenderMaster renMaster;
vtkCubeSource cubeSrc;
vtkPolyMapper cubeMap;
vtkActor cubel;

# create the rendering window and renderer
set renWin [renMaster MakeRenderWindow];
set ren1 [renWin MakeRenderer];

# connect the pieces and draw the result
cubeMap SetInput [cubeSrc GetOutput];
cubel SetMapper cubeMap;
$ren1 AddActor cubel;
$renWin Render;

```

Listing 3. A simple Tcl script using the wrapped Visualization Toolkit.

```

vtkContourFilter c;
c SetEndExecuteFunction {puts "Done"}

```

4. Results

Using the combination of techniques described above we have developed an automated technique for wrapping a C++ library into Tcl. We have used this for over a year in developing The Visualization Toolkit which now contains over three hundred classes and thousands of methods. The process has required almost no human interaction. The most common issue is adding a few methods to the hint file now and then. We managed to do this with making only a couple changes to our code. There were a few classes that had helper classes defined in the same header file. To handle this we added a special comment as below:

```

// BTX - short for begin Tcl exclude
// code that breaks our parser in here
// ETX - short for end Tcl exclude

```

Taking advantage of the dynamic loading features of Tcl7.5 we have wrapped additional C++ libraries into Tcl so that they can be loaded at run time by the scripts that require them. At this point our automated technique wraps about 98% of the possible methods within VTK and generates about 100,000 lines of source code. We have had success applying the same approach to wrap VTK into Java, although some of the issues are different. This ability to generate wrapper code for Tcl or Java is a nice feature.

Unfortunately this technique does not leverage any of the recent object oriented extensions to Tcl (such as

Otcl) and it is not a general purpose technique that can wrap any C++ code. As nice as that would be, it was through our moderated use of C++ that we were able to successfully wrap the Visualization Toolkit and a few additional C++ libraries into Tcl. In the future it might be worth considering using our approach to parse the C++ header files and then generate CDL files for Otcl. This would remove one of the primary obstacles to our use of Otcl.

Listing 3, shows a simple example of using the C++ Visualization Toolkit from within Tcl. More information on The Visualization Toolkit can be found at <http://www.cs.rpi.edu/~martink>. While our wrapper generator isn't suitable for many C++ class libraries, you can obtain a copy of it by emailing a request to martink@crd.ge.com.

I would like to acknowledge Bill Lorensen and Will Schroeder for their suggestions, encouragement and help in this work.

References

- [1] P. Bogdanovich. "Objective-Tcl: An Object Oriented Tcl Environment" *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.
- [2] W. E. Lorensen, B. Yamrom. "Object Oriented Computer Animation." *Proceedings of IEEE NAECON*, 2:588-595, Dayton Ohio, May 1989.
- [3] M. J. McLennan., "[incr Tcl]: Object-Oriented Programming in Tcl" *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[4] M. J. McLennan, "The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More" *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.

[5] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[6] D. Sheehan. "Interpreted C++, Object Oriented Tcl, What next?" *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.

[7] D. Wetherall, C. J. Lindblad. "Extending Tcl for Dynamic Object-Oriented Programming" *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.

Tksh: A Tcl Library for KornShell

Jeffrey Korn
Princeton University
35 Olden Street
Princeton, NJ 08544
jlk@cs.princeton.edu

Abstract

This paper describes Tksh, an implementation of the Tcl C library written on top of the library for the new KornShell (ksh93). Tksh emulates the behavior of Tcl by using the API that is provided for extending ksh93, which is similar to the Tcl library in that it allows access to variables, functions and other state of the interpreter. This implementation requires no modification to ksh93, and allows Tcl libraries such as Tk to run on top of ksh93 unchanged, making it possible to use shell scripts in place of Tcl scripts. ksh93 is well suited for use with Tk because it is backward compatible with sh, making it both easy to learn and easy to extend existing scripts to provide a graphical user interface. Tksh is not yet another port of Tk to another language – it allows Tcl scripts to run without modification using the ksh93 internals. This makes it possible to combine Tcl and ksh93, which is useful for writing ksh93 scripts that use components that have been implemented in Tcl (such as Tk widgets).

1 Introduction

Tcl[9] is a general-purpose scripting language which can be reused for a variety of different tools. It is designed to be extensible by allowing operations to be performed through a C library interface, such as adding new commands and manipulating variables. Programs such as *expect*[8] and *Tcl-DP* have been developed this way. The most well known and one of the most useful extensions to Tcl is Tk, which allows graphical user interfaces to be developed quickly and easily.

One of the reasons that Tcl/Tk is widely used is that developing user interfaces with high-level scripting languages can often be easier and faster than using a low-level language such as C or C++. High-level languages are able to hide many bothersome details such as storage management from the user.

Tcl/Tk is not the only user interface scripting language available for UNIX systems. The Desktop KornShell (dtksh, formerly wksh), which is part of the Common Desktop Environment (CDE), is one alternative. The CDE is being (or will be) shipped as a standard part of most UNIX systems. dtksh allows users to develop Motif user interfaces using the new KornShell command and programming language (ksh93)[2]. Several reasons make KornShell a reasonable choice to be the underlying scripting language for CDE:

1. *Backward Compatibility* - A large number of users have programmed in either the Bourne Shell or KornShell. It is possible to write scripts without having to learn a new language. Furthermore, existing shell scripts can be easily adapted to provide a user interface.
2. *Conformance to Standards* - KornShell conforms to the IEEE P1003.2 and ISO 9945-2 shell standards[13]. It deals with the issues of internationalization such as error messages, collation order, and international character classes used in pattern matching.
3. *KornShell is a Powerful Language* - ksh93 has a full set of programming constructs such as functions, floating-point math, associative arrays, loops, and pattern matching. With such a rich set of features, having to spawn other processes is infrequent, making ksh93's performance good.
4. *Additional Features* - ksh93 has additional powerful features that are not typically found in other scripting languages. ksh93 is good for interactive use, with features such as command line editing and job control. It also has features that make it easy to work with processes and files, yet is not tied to UNIX and runs on other operating systems.

Despite the strengths that dtksh has as a scripting language, users often prefer to use Tk rather than the Motif API for developing user interfaces. It can be harder to develop interfaces using dtksh because the set of graphics primitives included with dtksh is similar to the interface to Motif and X Intrinsics, which were designed to be used with a low level language like C. Since Tk is not designed around these interfaces, it is both easier to use and portable to other windowing environments (such as Windows and MacOS). With the growing popularity of Tk, there have been efforts to port Tk to other programming languages so users of these languages can take advantage of Tk's features. There are versions of Tk for Perl (TkPerl[1]), Scheme (STk[4]), Python (Tkinter[3]), and ML (CamlTk[12]). Tksh has come into existence because of the growing demand for a toolkit with the strengths of ksh93 as a scripting language coupled with the strengths that Tk has for building user interfaces.

The approach taken here is similar to the approach taken in STk; Tksh is not a modification to Tk (as is TkPerl). Instead, it attacks the problem at a lower level. Since Tk is written using the set of Tcl internals, Tksh is a layer in between Tk and ksh93 that provides the interface of the Tcl C API on top of the internals of ksh93.

However, Tksh is significantly different than other Tk ports in several ways. Tksh allows unmodified Tcl scripts to be evaluated in the KornShell interpreter, providing compatibility with the large number of existing Tcl/Tk scripts. Thus, transition from Tcl/Tk to Tksh is easier since scripts do not need to be rewritten. The execution of a Tcl script shares state with the ksh93 interpreter, which means that ksh93 and Tcl scripts can be used together, sharing the same function and name space. For example, a Tcl script can specify a ksh function to be executed as a callback for a Tk event. Similarly, a Tksh script can specify a Tcl procedure to be executed for a callback. It is therefore possible to use widgets written in Tcl (such as an open file dialog box) in a Tksh script.

Tksh can also be more convenient than Tcl/Tk for some tasks. For interactive scripting, features such as command line editing and job control are useful. Scripts that do a lot of work with processes and files are easy to construct with Tksh.

The rest of this paper is organized as follows: The next section introduces the fundamental concepts

underlying Tcl and ksh93, emphasizing the differences and similarities in the languages. Following that, the use and implementation of Tksh is described. Examples of applications using Tksh will be given. We conclude with information on future directions of this work.

1.1 Tcl and ksh93

Tcl is generally recognized as an effective scripting language suitable for writing useful programs. Tcl is often compared to languages such as Perl or Python, but is rarely compared to programming with shell due to the misconception that shells are inherently inadequate for writing large programs. This notion has developed partly because most of the widely used shells lack important features that are found in languages such as Perl and Tcl. Another reason for this opinion is that shells are traditionally slow because so much time is spent in the `fork` and `exec` system calls.

ksh93, the most recent version of the KornShell, offers many new features absent in traditional shells. Functionality is similar to Tcl, and spawning external programs is infrequently necessary resulting in increased performance (for example, command substitution for built-in commands does not create a separate process). Therefore, it is worthwhile to compare ksh93 to Tcl. This section gives an overview of some of the differences and similarities between the two languages.

C Library Interface

Both Tcl and ksh93 are built on top of a C library that consists of a set of functions which manipulate the state of the interpreter. This library is useful for embedding and extending the language. The two libraries provide roughly the same set of features (such functions to manipulate variables, add commands, use hash tables and dynamic strings, etc.), but provide different interfaces.

Interpreters

Tcl has the notion of an interpreter, which is a structure that maintains the state of a Tcl script such as defined commands and variables, and the execution stack. Although only one interpreter is used in most Tcl applications, it is possible to use several interpreters in a single program. ksh93 does not have the same notion of multiple interpreters, but does have the capability to create separate name and function spaces.

Commands

In Tcl, commands can be added to an interpreter using the function `Tcl_CreateCommand`. Once a command has been created, scripts that call the command with the given name will result in a call to the given function. ksh93 allows built-in commands to be added to the interpreter in a similar manner. In both languages, the function for created commands takes an `argv` list along with a pointer for private data. On architectures that support dynamically linked shared libraries, built-ins can be added at runtime by using the `builtin` command in ksh93. Although Tcl 7.4 does not support dynamic loading of built-ins, Tcl 7.5 has this feature.

Traces

A trace is a function that is associated with an action performed on a particular Tcl variable. Traces can be set for three types of events: (1) when the value of the variable is read from, (2) when the value of the variable is written to, and (3) when the variable is unset (destroyed). ksh93 has an equivalent notion called *disciplines*. Discipline functions can be C functions or shell functions and can be stacked so that multiple discipline functions apply to a variable. At the shell level, functions whose names are in the format *variable_name.action* are discipline functions. The discipline actions `get`, `set`, and `unset` are permitted for all variables, and other discipline names can be defined using C code extensions.

Results

Commands in Tcl produce two results. First, there is an integer completion code that indicates success (with `TCL_OK`) or failure (with `TCL_ERROR`). Second, there is a field in the interpreter structure that points to a result string. The result string may either be the result of the successful command or the error message generated by an unsuccessful one. ksh93 is similar, but uses standard output and standard error to return strings. This model makes it possible for shell functions and built-ins to behave like external UNIX commands.

Control Structures

Tcl implements control structures such as `if`, `while`, `for` and `foreach` as regular commands. They are not handled specially by the language, as Tcl consists of very few rules for parsing commands. ksh93 has the same set of control structures, but they are part of the language.

Name Space

Tcl has a single flat name space for each interpreter. Multiple name spaces can be introduced by creating multiple interpreters. ksh93 uses a hierarchical name space for variables with `.` as the separator. A compound assignment statement makes it easy to assign compound variables. For example,

```
point=(x=3 y=4)
```

assigns 3 to the variable `point.x` and 4 to the variable `point.y`. This method can be used to define data structures.

Variables

Both Tcl and ksh93 deal primarily with strings, and support associative arrays. Tcl allows a variable to indirectly reference another variable through the use of the `upvar` command. With ksh93, references can be defined by using a reference variable type. For example, defining `nameref foo=bar` will cause each subsequent operation on variable `foo` to be an operation on the variable `bar`. Reference variables make call-by-name reference possible without requiring the use of `eval`. ksh93 also has some additional variable types not present in Tcl. Floating point numbers are supported for performing arithmetic, so invoking commands such as `bc` and `awk` are no longer necessary. ksh93 also has indexed arrays in addition to associative arrays.

Expressions and Patterns

Both Tcl and ksh93 support C style expressions, including most arithmetic operators such as `<<` and `~`. With Tcl, the command `expr` is used to evaluate a string representing a C expression. In ksh93, C expressions may be used inside `((...))` notation. ksh93 also allows C style assignments such as `+=` in expressions.

Tcl uses an extended form of regular expressions for pattern matching. Although shells traditionally provide notation that is far more restrictive than regular expressions, ksh93 also uses an extended form of pattern matching that is very similar to Tcl. The pattern matcher in ksh93 includes a way to negate expressions, which is not currently possible with Tcl (for example, it is possible to invoke the command `ls !(*.o)` to list all files that don't end in `.o`).

Quoting

The quoting constructs in Tcl are designed to be easily nested. For example, `[...]` notation is used for

command substitution and `{...}` is used for literal quoting. KornShell behaves similarly for command substitution by introducing the `$(...)` syntax to replace the `'...'` syntax from Bourne shell. `ksh93` does not have a way to nest literal strings, but this is generally not necessary since nested evaluation is much less common in the shell. `ksh93` also allows ANSI C character sequences to be expanded with the `$'...'` syntax.

Performance

Although it is hard to precisely measure language performance, `ksh93` has been reported to be comparable to the speed of Perl[5]. Testing shows there is only a 20% CPU-time penalty from using `ksh93` as opposed to a totally C based application[11]. Another study[14] quotes the speed of `ksh` as being in between the speed of Tcl and Perl.

2 Using Tksh

Tksh implements the library interface for Tcl version 7.4, and works with Tk versions 3.6 and 4.0 (support for Tcl version 7.5 and Tk version 4.1 is in development). Tksh can be either statically linked with `ksh`, or used as a shared library. As a shared library, Tcl functionality can be loaded by using the `ksh93` builtin command to load the initialization command `tclinit` which initializes the Tcl library. Tk can be loaded similarly with the command `tkinit`.

Once Tk functionality is loaded, all of the the commands from Tk exist as shell builtins, and `ksh93` can accept Tk commands with `ksh` syntax. For example, the "Hello, world" program can be written as follows:

```
pack $(message .b -text "Hello, world")
```

Scripts can be written that take advantage of features in `ksh93`. For example, the following command can be used in an interactive shell to create a button that always displays the current directory. When the button is clicked, the current directory will be set to the user's home directory:

```
pack $(button .b -textvar PWD -command cd)
```

When using Tksh, all Tcl commands are available and can be used by prepending `tcl_` to the corresponding Tcl command. For example, the Tcl command `info` has the name `tcl_info` in `ksh93`. The exact Tcl names are not used because many of the commands have conflicting names with `ksh93` and UNIX commands. For example, Tcl contains commands such as `while` and `if`, which conflict with

the corresponding keywords in `ksh93`. Although using Tcl commands in a `ksh93` script is infrequently necessary, doing so can be more convenient for a programmer familiar with Tcl. The following example uses the Tcl command `trace` to create a window that always displays the contents of the current directory:

```
pack $(listbox .list -width 20)
tcl_trace var PWD w trace_pwd
function trace_pwd
{
    .list delete 0 end
    .list insert end *
}
```

A discipline could be used instead of a trace in the above example by eliminating the `tcl_trace` command and changing the name of `trace_pwd` to `PWD.set`.

Figure 1 on the next page has the listing of a small Tksh program. The script displays a window containing the contents of the current directory as well as a list of visited directories in an interactive shell. Double clicking on the name of a file will invoke the editor on the selection; double clicking on a directory will set the current directory to the selection. If the a command is typed into the shell that causes the current directory to change, the window is updated. Sample output is shown in Figure 2.

As the listing of the program indicates, code to set up a user interface with Tksh tends to look similar to Tcl scripts with different quoting rules. The parts that differ the most are in the implementation of the script's functionality.

Evaluation of Tcl Scripts

Tcl code can be embedded into a `ksh93` script by using the `source` built-in command. By issuing the command `source [filename]`, the file `filename` (or standard input if no file is specified) is parsed and interpreted as a Tcl script. The sourced Tcl script has access to and can change the current state of the `ksh93` interpreter.

Most existing Tcl/Tk scripts can be evaluated using `source` with no modification. For example, the widget demonstration that comes with Tk works without having to make any changes. Because Tksh can interpret Tcl, it is able to work with the the large number of existing Tcl/Tk scripts. Thus, an application can be built with Tksh where the main functionality is written in `ksh93`, and widgets are used

```

function selectFile
{
    [[ -d "$1" ]] && cd "$1" > /dev/null
    [[ -f "$1" ]] && ${EDITOR-${VISUAL-emacs}} "$1"
}

function PWD.set          # Discipline called when directory changed
{
    nameref dir=.sh.value      # .sh.value is value being stored
    .f.ls.list delete 0 end
    .f.ls.list insert end .. *
    [[ ${VisitedDir["$dir"]} != "" ]] && return
    .f.dirs.list insert end "$dir"
    VisitedDir["$dir"]=1
}

typeset -A VisitedDir      # Associative array
set -o markdirs

pack $(frame .f)
pack $(frame .f.dirname -relief raised -bd 1) -side top -fill x
pack $(frame .f.ls) $(frame .f.dirs) -side left
label .f.dirname.label -text "Current directory: "
label .f.dirname.pwd -textvariable PWD
pack .f.dirname.label .f.dirname.pwd -side left

scrollbar .f.ls.scroll -command ".f.ls.list yview"
listbox .f.ls.list -yscroll ".f.ls.scroll set" -width 20 -setgrid 1
pack $(label .f.ls.label -text "Directory Contents") -side top
pack .f.ls.list .f.ls.scroll -side left -fill y -expand 1

scrollbar .f.dirs.scroll -command ".f.dirs.list yview"
listbox .f.dirs.list -yscroll ".f.dirs.scroll set" -width 20 -setgrid 1
pack $(label .f.dirs.label -text "Visited Directories") -side top
pack .f.dirs.list .f.dirs.scroll -side left -fill y -expand 1
bind .f.dirs.list "<Double-1>" 'cd $(selection get)'
bind .f.ls.list "<Double-1>" 'selectFile $(selection get)'

cd .

```

Figure 1: Listing of Sample Program

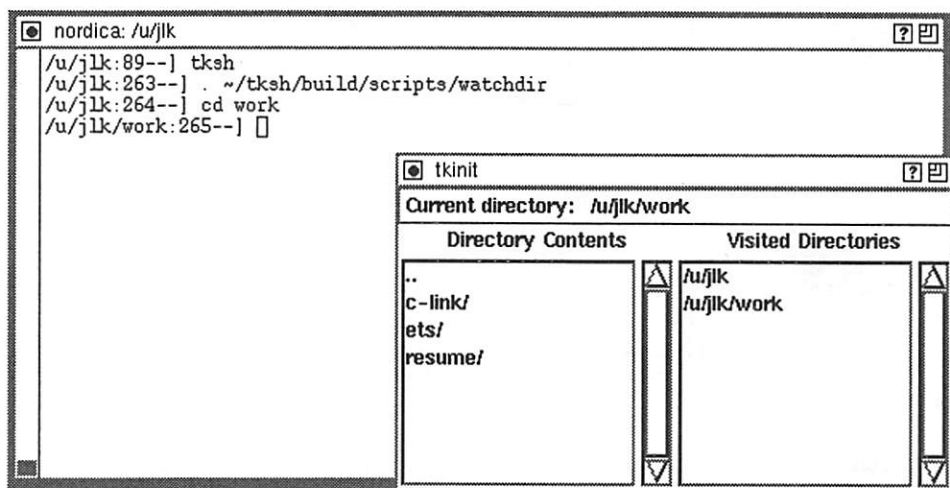


Figure 2: Screen Dump From Sample Program

that have already been written in Tcl. The transition from Tck/Tk to Tksh is therefore a smooth one. For example, if we were writing a Tksh script that required a dialog box to open file, we could take code directly from one of the many Tcl implementations rather than translate it or rewrite it from scratch.

Functions declared with `proc` will be parsed as Tcl source when they are subsequently called, even after `source` returns. For example, if we have the following script:

```
function foo
{
    X=37
    print "$(bar test)"
}

source <<'EOT'
proc bar {args} {
    global X
    set X [expr $X + 1]
    return "Proc bar: args: $args, X: $X"
}
EOT
```

A call to the function `foo` will result with the following output:

```
Proc bar: args: test, X: 38
```

Note that the Tcl procedure `bar` is able to use and modify the shell variable `X`. It is also possible for Tcl source to invoke ksh93 functions and built-ins.

Mixing Tcl and ksh93

Some Tk commands take arguments that are strings to be interpreted under certain conditions. For example, the `button` command allows a string to be

specified (with the `-command` option) that will be executed when the button is pressed. A problem arises with this situation because Tksh needs to decide whether the string should be parsed as a Tcl command or a shell command. Normally, Tksh will use the current mode of the parser, but this is not always desirable. Therefore, Tksh has introduced a special notation to specify the language of the command string.

If a command string has as its first line `#!ksh`, ksh will be used, and if the first line is `#!tcl`, Tcl will be used (otherwise the current parser is used). Tksh augments the standard `bind` command in Tk (which binds a command string to a window event) with a wrapper function that automatically places the appropriate line to the beginning of the specified string (if one hasn't already been specified).

Lists

Tcl uses lists to deal with a collection of strings, whereas ksh usually uses arrays. Thus, Tcl has a built-in set of functions for dealing with lists and ksh does not. However, ksh does have the facilities that make it possible to provide lists. A list can be represented as a string if the elements of the list are quoted in such a way that splitting the list in ksh (by processing the arguments) yields the elements of the list. For example, the Tcl list `{a {b c} d}` corresponds to `'a "b c" d'` in ksh. If we were to invoke the command `eval set -- 'a "b c" d'`, `$1` would be `a`, `$2` would be `b c`, and `$3` would be `d`.

Tksh provides two different ways to deal with lists, which can be specified with the built-in command

`listmode`. The first way is to use ksh style lists when parsing ksh, and Tcl style lists when parsing Tcl (this is the default). The second way is to always use Tcl style lists. This mode is useful because some Tcl applications don't use the appropriate Tcl API to manipulate lists, and assume the syntax of the list instead (for example, a function would return `"{a {b c}}"` directly instead of using the C function `Tcl_Merge` to create the list from the individual elements.

A Tcl style list can be used in a shell script by using the built-in command `setlist`. Depending on the options, the `setlist` command will convert a Tcl style list into either a ksh style list, an array, or the positional parameters (`$1`, `$2`, etc.).

Tcl Result Strings

In Tcl, commands return both a string and a completion code. In ksh, commands return only a completion code. The means by which strings are returned in ksh is by printing them to standard output. Thus, Tksh prints the contents of the result string to standard output upon successful completion of a Tcl command, and prints the result string to standard error otherwise. This allows command substitution in ksh behave like it does in Tcl. Since Tcl does not normally print the result of a command, Tksh only prints the result of a Tcl command if it is called inside command substitution.

One important difference between command substitution in Tcl versus ksh has to do with side effects. Command substitution in ksh93 behaves as if the command was executed in a separate process, which means that anything done inside substitution cannot change the state of the interpreter (the only exception to this is that built-in commands can be added). However, this is not the case with Tcl. Although not generally a problem, this occasionally leads to some unexpected behavior when substituting a Tcl command within a ksh command. Future versions of ksh93 are expected to have an option to allow side effects in command substitution.

3 Tksh Architecture

Tksh is written in C and consists of around 5,000 lines of source code along with a few modules that have been taken directly from the source of Tcl. Most of the code is straightforward and maps the semantics of a Tcl library function to the corresponding ksh93 call (if such a function exists). In several

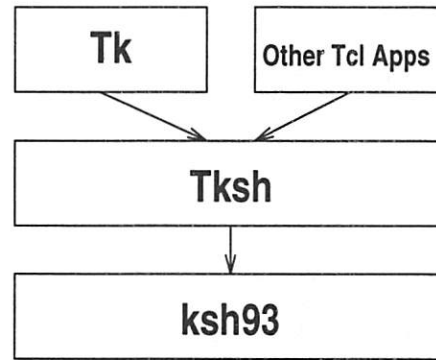


Figure 3: Tksh Architecture

cases, there is either no corresponding function in the API, or the semantics of the similar functions differ in a nontrivial way. This requires writing code to emulate the correct semantics. Examples of this are described throughout this section.

The parts of code that have been taken from Tcl without requiring modification include the hash table and dynamic string libraries, the Tcl parser, the implementation of most of the builtin commands (such as `proc` and `set`), and a few miscellaneous routines. While ksh93 has a hash table and string library, it is easier to use Tcl's routines for the moment because semantics of the libraries differ.

Variables and Traces

Traces for variables are implemented on top of the discipline mechanism of ksh93. Although Tcl traces are similar to disciplines, there are a number of subtle differences in the mechanisms. For instance, when a Tcl trace is invoked on a variable, all other traces for that variable are turned off for the duration of the trace. ksh93, on the other hand, allows other disciplines to execute as long as they are further down on the discipline stack. Although it would have been possible to implement a separate tracing facility that was called from the `Tcl_GetVar` and `Tcl_SetVar` functions, using ksh93 disciplines is advantageous for several reasons. If a variable is read from, written to or unset from a shell script, `Tcl_GetVar` and `Tcl_SetVar` are never called (and thus the traces wouldn't be). However, if the traces are installed as disciplines, they will be called in this case. Also, this implementation allows traces and disciplines to be interleaved on the same variable – a trace can be followed by a discipline which is followed by another trace.

Evaluation

In order to interpret Tcl scripts in ksh, an internal flag has been added to Tksh that indicates which language is currently being interpreted. If we are interpreting ksh code, the `Tcl_Eval` and `Tcl_EvalFile` routines effectively hand off the string to be evaluated to the ksh93 equivalent, `sh_eval`. If we are interpreting Tcl code, the Tcl parsing routines are called instead, which generate an `argv` list to be evaluated. Before the `argv` list can then be handed off to `sh_eval`, command names are mapped appropriately – Tcl command names such as `set` and `proc` are mapped to names that won't conflict with ksh93 (`tcl_set` and `tcl_proc`).

The Tksh built-in command `source` sets the aforementioned flag to indicate Tcl code is to be evaluated, and calls `Tcl_EvalFile` with the argument. When `source` has completed, the flag is restored to its original state.

Another modification has been made to the way the Tcl command `proc` behaves. Normally, the implementation of the `proc` command creates a Tcl command with the given name that corresponds to a C function named `InterpProc`. When `InterpProc` is called, it passes a field of its private data, which contains the string of the function body, to `Tcl_Eval`. In Tksh, `InterpProc` first sets the flag indicating Tcl code is to be evaluated before calling `Tcl_Eval` on the function body. This way, a Tcl function can be called at any time (in ksh or Tcl code), not just within the `source` command.

Dynamic Scope

One important difference between ksh93 and Tcl is that Tcl uses dynamic scoping, while ksh93 uses static scoping (Tcl uses dynamic scoping in order to access an array defined in an enclosing function; ksh93 uses reference variables to accomplish this). The Tcl commands `upvar` and `uplevel` make use of such dynamic scoping, and have been rewritten in Tksh. Although the stack of enclosing scopes is inaccessible at the scripting level of ksh93, they are accessible through the C language API. Tksh uses these functions to achieve the correct behavior.

Files and Processes

Tcl has a set of commands that deal with files and processes. This part of Tcl has been rewritten in Tksh to make use of ksh93's handling of processes and files. In Tcl, a function called

`Tcl_CreatePipeline` exists to start a set of processes and communicate with them (this library function is used by Tcl's `exec` command). Tksh implements this function by putting together an equivalent ksh command string from the arguments and evaluating it in the shell. For example, the Tcl command:

```
exec cat << "Hello" | tee foo >>stderr
```

would be translated to the following shell command:

```
cat <<HUP | tee foo 1>&2
Hello
HUP
```

In shell, only 10 files can be referred to at a time (by specifying a number from 0 to 9 in front of a redirection symbol). In Tcl, the number of files that can be specified at once is bound by the process limit and thus it is possible to specify more files at once in Tcl scripts. Tksh gets around this problem by mapping as many Tcl files to descriptors 0 through 9 as possible. If a file is not mapped, shell descriptors are moved around to map the file before the command is invoked, and restored afterwards. This implementation has the advantage that it allows files to be shared by ksh93 scripts and Tcl scripts, since ksh93's underlying file facilities are used for both.

Events

In Tcl/Tk, execution is driven by an event loop. Tk has a mechanism which is used to add events which will invoke a given procedure when the event occurs. ksh93, on the other hand, has its own event mechanism that is part of the shell (the newest releases of Tcl/Tk have moved the event loop from Tk into Tcl).

The library for ksh93 contains a routine called `sh_waitnotify` which is used to specify a function to be called when the shell is waiting for either input or a child process to complete. The notify function should return when input is ready or when a signal is received (it must also return within timeout, which is specified). Tksh uses the `sh_waitnotify` mechanism to run the Tk event loop. It registers an event for the file descriptor that the shell is waiting on, as well as an event to occur after the timeout period. The function `Tk_DoOneEvent` is repeatedly called until such an event occurs. Signal handlers are also placed to trigger an event when a relevant signal occurs.

The event mechanism in ksh93 has a major advantage over Tcl's. If a command is executed, ksh93

will process events while waiting for the command to complete. Tcl, on the other hand, will freeze until the child process terminates. ksh93 is also able to parse its input as it is entered with this mechanism since it owns the event loop.

Parsing of Tcl Code

The code for Tcl is structured so well that it was possible to place Tcl's parser directly into Tksh without modification. The Tcl parser goes through a string and converts it into an argument list, performing any necessary variable or command substitutions on the way. Variable substitutions are done by using the library function `Tcl_GetVar`, which (as mentioned previously) has been rewritten to use the ksh93 API. Command substitutions are done by calling `Tcl_Eval` which recursively calls the parser.

The function `Tcl_Eval` calls the parser to generate the argument list, and then looks up the procedure to be invoked in the ksh93's built-in command table. If the command exists, it is executed with the argument list.

Patterns and Lists

The ksh93 API has a routine which converts a regular expression into a shell pattern. Shell patterns are more powerful than regular expressions, and the translation from regular expressions is fairly simple. The Tcl regular expression functions are implemented by first translating regular expressions into shell patterns, and then calling the appropriate ksh93 library function.

A string is converted into a ksh93-style list element using the function `sh_fmtq`, which returns a shell-quoted version of its input string. A list is thus created by concatenating quoted versions of each element.

Tcl Commands

Most Tcl commands that deal with internals of the interpreter are implemented using the API of the Tcl C library. As a result, such commands work with Tksh without modification. For example, the implementation of the `set` command is shown in Figure 4. The command makes use of `Tcl_SetVar`, `Tcl_GetVar`, and `Tcl_AppendResult`. Since each of these functions are implemented as part of Tksh, there is no need to make modifications to the `set` command itself. There are, however, a handful of Tcl commands that use functions which are not part

```
int Tcl_SetCmd(dummy, interp, argc, argv)
ClientData dummy;
register Tcl_Interp *interp;
int argc;
char **argv;
{
    if (argc == 2) {
        char *value;
        value = Tcl_GetVar(interp,
                           argv[1], TCL_LEAVE_ERR_MSG);
        if (value == NULL) {
            return TCL_ERROR;
        }
        interp->result = value;
        return TCL_OK;
    } else if (argc == 3) {
        char *result;
        result = Tcl_SetVar(interp, argv[1],
                           argv[2], TCL_LEAVE_ERR_MSG);
        if (result == NULL) {
            return TCL_ERROR;
        }
        interp->result = result;
        return TCL_OK;
    } else {
        Tcl_AppendResult(interp,
                         "wrong # args: should be \"",
                         argv[0], " varName ?newValue?\"",
                         (char *) NULL);
        return TCL_ERROR;
    }
}
```

Figure 4: Listing of `Tcl_SetCmd`

of the exported interface. Commands such as `info` and `rename` are examples, and have been reimplemented for use with Tksh.

4 Current Work

Tksh is currently being used as a foundation to write a graphical debugger for C called TkCdb. The debugger runs by spawning an execution of a line oriented debugger such as `gdb` or `dbx` and communicating with the process through a pipe. The debugger uses external programs, such as a graph drawing program (called `dotty`[7]) to display data structures. Shell scripts can be specified to be executed at break-points as well.

Tksh is used as the debugging language because of its strengths as an interactive command language. Debuggers are interactive programs because commands are usually entered one at a time to a prompt

rather than being placed into a batch script. TkCdb takes advantage of the interactive facilities of ksh93 such as command line editing and job control. Using the command line interface to the debugger feels like being in the shell because it actually *is* a shell.

Although TkCdb has a rich set of graphical debugging features, it is less than 1,500 lines of Tksh. The simplicity, which can be attributed to using a high-level language and a powerful graphical toolkit, makes the debugger easy to understand, modify and extend. Future plans are to modify the C debugger to create a Tcl and ksh debugger.

Another project that is under development which uses Tksh is a file browser, similar to Window's Program Manager and the Finder in MacOS. The file browser is unique in that it is designed to work with an interactive shell; commands can be typed into the ksh command prompt that affect the browser, and actions triggered through the browser interface can affect the shell. For instance, the user can scroll through a list of the command history and double click on an entry to repeat that command.

5 Future Work

Support for Tcl 7.5

A significant number of changes have been made in the newest release of Tcl, version 7.5. These new features are currently being integrated into Tksh. One of the features introduced in Tcl 7.5 is a new I/O system, which is needed to support versions of Tcl on MacOS and Windows. The I/O system will be implemented for Tksh on top of the I/O system that is part of ksh93, called *sfio*[6]. *sfio* is a portable I/O library that is backward compatible with the C stdio library. It has functionality similar to the I/O system in Tcl 7.5, including the ability to expand the system for new file types and the ability to do nonblocking I/O. The fact that *sfio* is backward compatible with stdio is a major advantage.

Tcl 7.5 also provides facilities for loading dynamic libraries. ksh93 has this as well, so the proper interface should not be difficult to construct. Another major Tcl 7.5 feature is the ability to use multiple interpreters. This feature is also being developed in Tksh.

Once the support for Tcl 7.5 is complete, Tksh will be tested on non-UNIX platforms, such as Windows 95. Since Tksh has no system dependencies (the only

dependencies are on the ksh93 API), it should work on any system that runs both Tk and ksh93. This includes most UNIX compatible systems, Windows NT and Windows 95.

Performance

Performance has not been measured yet with any precision. At this stage, focus is more on functionality than speed. However, when running the set of included Tk demos on top of Tksh, performance difference is not perceptible. The same is true when running Tcl scripts. One area in which performance could make a difference is in the `Tcl_GetVar` and `Tcl_SetVar` functions. Other aspects, such as the parsing of Tcl strings, will not be affected because the actual Tcl code is being used. Future work will include finding parts of Tksh in which performance can be improved.

Interface to Tk

Tksh currently provides the same interface to Tk that Tcl does. However, the use of disciplines in ksh93 makes it possible to provide a more object-oriented interface as well. For instance, suppose we wish to associate a function with a button that will be executed when the button is pressed. Currently, this might be implemented as follows:

```
button .b -text "Click here" -command foo

function foo
{
    ...
}
```

Using disciplines, we could do something like the following:

```
button .b
.b[text]="Click here"
function .b.mouseup
{
    ...
}
```

6 Summary

KornShell has many features that make it a suitable choice for writing scripts. It has functionality similar to both Tcl and Perl, good performance, and syntax that is less complex than Perl yet requires fewer levels of nested evaluation than Tcl does. It is also compatible with the Bourne Shell and conforms to

POSIX and international standards. The creation of an interface to the internals of ksh93 which is equivalent to the interface to Tcl internals allows ksh93 to be used with existing Tcl applications. The most notable of these applications is Tk. In addition, since Tksh can interpret Tcl scripts, existing Tcl/Tk scripts can be used with Tksh.

7 Availability

Tcl and Tk are available via anonymous ftp at [ftp.smli.com](ftp://smli.com) in the directory `/pub/tcl`. Source to ksh93 is freely available for academic purposes (from the author), and binaries are free for non-commercial use. Binaries can be obtained through the World Wide Web at <http://www.research.att.com/orgs/ssr/book/reuse>. For information on obtaining a copy of Tksh, see the web page <http://www.cs.princeton.edu/~jlk/tksh>.

References

- [1] M. Beattie, "TkPerl - A port of the Tk toolkit to Perl5", *Very High Level Languages Proceedings*, USENIX, 1994.
- [2] M. Bolsky, D. Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.
- [3] M. Conway, *A Tkinter Life Preserver*, <ftp://ftp.python.org/pub/python/doc/tkinter-doc.tar.gz>, 1994.
- [4] E. Gallesio, "Embedding a Scheme Interpreter in the Tk Toolkit", *Proceedings of the Tcl/Tk 1993 Workshop*, USENIX, 1993.
- [5] D. Korn, "ksh: An Extensible High Level Language", *Very High Level Languages Proceedings*, USENIX, 1994.
- [6] D. Korn, P. Vo, "Sfio: Safe/Fast String/File IO", *Proceedings of Summer USENIX Conference*, USENIX, 1991.
- [7] B. Krishnamurthy, *Practical Reliable UNIX Software*, Wiley, 1995.
- [8] D. Libes, *Exploring Expect*, O'Reilly, 1994.
- [9] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
- [10] J. S. Pendergrast, *Desktop KornShell Graphical Programming*, Addison-Wesley, 1995.
- [11] J. S. Pendergrast, *Answers to FAQ Questions*, <http://landru.unx.com/~pend/dtksh.html>, 1996.
- [12] F. Pessaux, F. Rouaix, *The CamlTk Interface*, <ftp://ftp.inria.fr/lang/caml-light/camltk.dvi.tar.gz>, 1995.
- [13] *POSIX - Part 2: Shell and Utilities*, IEEE Standard 1003.2, ISO/IEC 9945-2, IEEE, 1993.
- [14] S. Raney, "A comparison of Tcl/Tk, the Desktop KornShell and MetaCard", <http://www.metacard.com>, 1996.
- [15] B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, 1995.

SurfIt! - A WWW Browser

Steve Ball

PASTIME Project

Cooperative Research Centre for Advanced Computational Systems

Australian National University

ACTON 0200 Australia

Tel: +61 6 249 5146

Fax: +61 6 249 0010

E-Mail: Steve.Ball@surfit.anu.edu.au

Abstract

World Wide Web (WWW) browsers have, until recently, only offered static features and media types. A new generation of WWW browsers now becoming available are offering technologies to remove these limitations and allow dynamic extensions to the browser application as well as dynamic media types or active message content.

SurfIt! is an example of these “next generation” WWW browsers. It offers safe execution of downloaded Tcl/Tk applets¹ to support active message content, as well as hypertools to dynamically extend the browser at run-time.

The latest version of SurfIt! is written entirely as Tcl script code, making it highly portable, user-customisable and extensible

Keywords: World Wide Web, WWW, browser, Safe-Tcl, applets.

Introduction

SurfIt! is a new World Wide Web user agent (aka browser) which has been implemented entirely using Tcl/Tk script code. It supports most of the features usually found in Web browsers today - document retrieval using the HTTP protocol, graphical display of HTML documents, inline graphics, hypertext navigation, and so on. SurfIt! also includes many standard browser features such as local caching of documents, incremental document loading and display and concurrent document downloading. As a Tk application, it is easy to construct applications which can communicate with SurfIt!, using the send command, to implement the concept of hypertools. SurfIt! provides direct support for creating tools such as these.

However, the most interesting aspect of SurfIt! is its ability to execute any Tcl/Tk script, which may be downloaded from a remote server, within the context of the browser. These mini-applications, or “applets”, are evaluated in a separate, safe interpreter to ensure that they do not conflict with any other Tcl

¹ a “mini-application”.

code and that they cannot damage or compromise the user's computing environment in any way.

This paper discusses the implementation of the SurfIt! World Wide Web browser, including the motivation behind its development and a brief synopsis of its early history. Later sections describe the functionality offered by the browser and its internal architecture. Issues concerning the handling of Tcl applets and hypertools will be discussed and finally future goals will be outlined.

Motivation and History

The development of SurfIt! began in early 1995. While developing advanced Web-based hypermedia datasets it was found that active message content was needed to deliver the required user interface functionality. Also, there was a requirement to develop a continuous media² playback system which was tightly integrated with a Web browser, and yet loosely coupled [Ball95].

A possible solution to these requirements was to use Tcl/Tk to implement an active message content system as well as taking advantage of Tk's send facility to implement hypertools for handling continuous media. There were several problems facing this approach. The only Tk-based WWW browser then available for displaying textual content was tkWWW v0.12 [2]. This version of tkWWW was based upon Tk 3.6 and so it could not display inline graphic images, which makes it obsolete when compared to modern WWW browsers. Although tkWWW has the capability of executing downloaded Tcl scripts, it lacked the security features of Safe-Tcl to ensure system security. Another alternative would be to use WebRunner (now Hot Java), which used liveOAK (now Java) as its programming language. It was decided not to use this system since at that time the windowing toolkit available with liveOAK lacked many necessary features found in Tk

To address these problems, in February 1995 I created a prototype Web browser, TkWeb. Similarly to tkWWW, TkWeb used the CERN Common Code Library (libwww) to provide network protocol and

content type handlers, in particular a HTML parser. This parser was customised so that it would create a list of Tcl procedure calls. The Tk application supplied the procedure definition. When the libwww HTML parser parsed a document, it would pass the generated script to the Tk application which would then evaluate it. This resulted in the application-supplied procedures being invoked to render the document. I found integrating the CERN Common Code Library into a Tk application to be very difficult.

In April 1995 Stephen Uhler [Uhler95] implemented Hippo³: a WWW browser mostly written as a Tcl (version 7.4) script. At that time Tcl lacked network connectivity primitives, so Hippo used a HTTP protocol handler written in C. Hippo introduced the now infamous 8 line Tcl HTML parser. This parser filters HTML markup into a Tcl script which is then evaluated by the application to render the document. Again, the parser generates a script with calls to procedures that the application is expected to supply. Stephen released the HTML parser/renderer publicly circa May 1995 as `html_library`.

In late April/early May 1995 I received a copy of Stephen's `html_library` package as well as Jacob Levy's [Levy95] `stcl` package - a Safe-Tcl extension to Tcl version 7.4. Given the difficulties I encountered in attempting to make use of the CERN Common Code Library and that a pure-Tcl HTML parsing/rendering subsystem was now available I decided to implement a new Web browser completely in Tcl script code. The only missing functionality at that time was network connectivity, so the popular Extended Tcl package was used to provide low-level network access (another possibility would have been to use Tcl-DP). In addition, the table geometry manager from the BLT distribution was used to implement HTML tables.

Browser Features

The latest version of SurfIt!, version 0.5 [Ball96], is based upon Tcl version 7.5 and Tk 4.1. Since this

² time based media, such as audio or video

³ short for "Hypocratic"

version of Tcl/Tk includes network socket support and the grid geometry manager there is currently no requirement for any other extensions to be installed.

Below is a brief list of the major features of SurfIt.

- HTML v2.0 (RFC 1866) compliant.
- partial implementation of HTML v3.2 [Raggett96c].
- partial implementation of TABLE draft specification [Raggett96a].
- GIF, PPM and X bitmap graphics formats supported and may be inlined. Support for JPEG graphics format has been prototyped, but requires a robust photo image type format handler.
- Network protocol handlers for HTTP v1.0 and FTP. file: and mailto: URLs are also supported.
- Asynchronous, concurrent document downloading with incremental document display.
- .mailcap processing for "Helper Applications".

User Interface

The primary focus of a Web browser is the display of a hyperdocument (which may include embedded documents). Each hyperdocument is displayed in a separate hyperwindow. There will be various functions that may be performed upon hyperdocuments, for example scrolling or activating hyperlinks, and functions for hyperwindows, for example loading new documents, history navigation, controlling applets and so on. The browser's user interface must provide controls for all of these functions. In addition there are functions that are not related to any hyperwindow, and these are termed browser-level functions. Browser-level functions include creating new hyperwindows, manipulating the local cache, setting user preferences and so on. SurfIt! explicitly separates the user interface for hyperdocuments and hyperwindows from the interface for browser-level functions by placing only browser-level controls in the main window, and each hyperwindow is placed in a toplevel widget. This

approach eliminates redundancy in the user interface when the user has more than one hyperwindow open.

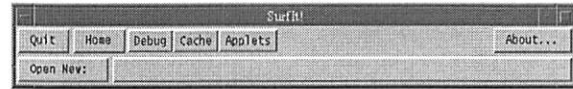


Figure 1: The SurfIt! main window.

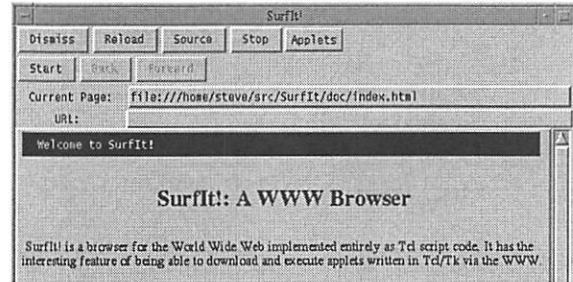


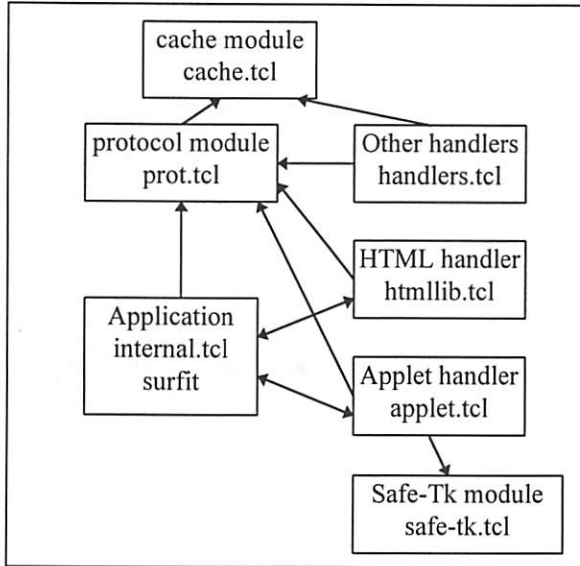
Figure 2: A hyperwindow

Since SurfIt! supports the execution of untrusted applets, and it allows those applets access to the Tk Text widget of the hyperdocument into which they were loaded, it is of vital importance to provide a user interface where there is always a means by which the user can control all applets. Safe-Tk, described below, restricts an applet's use of Tk to a widget sub-heirachy rooted at the hyperdocument's Text widget or a separate toplevel widget. The applet is quite at liberty to compromise any widget to which it has access, and so may disrupt the proper functioning of the browser with respect to that hyperdocument. If the applet is deemed to be behaving inappropriately by the user then by virtue of the design of the browser user interface they may terminate the applet, destroy the hyperwindow and initialise a new hyperwindow as necessary. Hence proper functioning of the browser may be restored.

Architecture

The various modules which underly SurfIt! have been designed with reuse as a goal. Other applications are required to access Web documents, for example a web crawler, and these applications are more easily implemented if given access to SurfIt!'s lower-level functionality.

Below is an overview of SurfIt!'s module structure.



The Protocol Module

The central module is the Protocol module. It manages the loading of documents and their handling by content-type handlers.

When building a World Wide Web user agent, it is a mistake to take the myopic view of Web documents as only being in HTML format, and delivered to the browser using only the HTTP network protocol. In fact, the Web subsumes the FTP and Gopher network protocols for document retrieval, and in addition documents may be retrieved from the local filesystem or by other means - not just HTTP. So the browser developer is faced with the questions: What is a document? How is a document's data delivered to the browser?

A document may be viewed as an atomic data object. Documents may have many media types: plain text, HTML marked-up text, JPEG graphics, MPEG movies, and so on. It is important to note that all of these types of documents are first-class documents and must be able to be loaded as a hyperdocument - they do not have to be embedded in an HTML document. Also, MIME encoding allows a single message to contain several documents (a multi-part message), so the developer should not assume that a message or file is equivalent to a document.

The key issue in loading a document is that the WWW provides a uniform mechanism for specifying

a document's location - its Uniform Resource Locator, or URL. An URL has two parts, separated by a colon: the protocol and then the document specifier, the interpretation of which is dependent on the given protocol. For example:

`http://surfit.anu.edu.au/Surfit/`
 protocol host:port path
`mailto:Steve.Ball@surfit.anu.edu.au`

SurfIt! presents a simple programming interface to a generalised mechanism for dealing with a multiplicity of protocol handlers and document content-type handlers. The Protocol module allows handlers to register themselves and manages the calling of the appropriate handler in response to a request to load a document. The registration system allows new handlers to be added at run-time. The application requests an URL to be loaded using the PRloadDocument procedure. A unique global variable is created to contain relevant information for loading the document. The name of this variable is passed to most of the procedures which subsequently deal with the document. A document's data may be presented to the application in several ways and these are indicated by an enumerated type:

- PRdata the data is a Tcl string.
- PRfile the data is in a file in the local filesystem.
- PRfd the data is to be read from a network channel.
- PRpost the data has been post processed by a content-type handler.

The Protocol module uses fileevent scripts to allow documents to be downloaded concurrently. The fileevent script invokes procedures which manage the process of reading data from a channel, passing that data to the content-type specific processor and rendering procedures, as well as taking of other housekeeping such as cache management.

A protocol handler has two parts (procedures). The handler itself, which is registered with the Protocol module, and a read handler. When an URL specifying that protocol is requested the handler is

invoked. The handler commences the document load, and defines the read handler to be used to retrieve the actual data. The read handler is called to read the data of the document as it becomes available, which it returns as its result. A read handler may also prepend data which has been pushed back onto the data stream by a content-type handler. The read handler always writes the data into the cache, which is the only way to handle binary data. In the case of supposedly text documents the newly retrieved data is read back from the cache file to be returned as the procedure result.

Simultaneously downloading documents, and their rendering onto the display, would naturally be done using threads. Unfortunately, Tcl does not provide multithreading. However, concurrent downloading and rendering using fileevent scripts has proven quite effective for incremental document display. Most other modern Web browsers use the same technique, with the Hot Java browser being the only obvious exception.

The Protocol module supplied with Surft! includes HTTP and FTP network protocol handlers, as well as a handler for the file: protocol which accesses the local filesystem. A mailto: protocol handler is supplied with Surft! in a separate module.

A content-type handler also has two parts: a document processor and a document renderer. The processor procedure is invoked to process the raw data of the document. The result returned by the processor then becomes the input of the handler's renderer procedure. A processor can also register its output with the Cache module as post-processed data. When a document is available in a post-processed form the data is passed directly to the renderer procedure, thus saving some processing overheads. Examples of post-processed data include images, where the data is read into a Tk image, and HTML documents, where the processor procedure filters the HTML data into Tcl commands which the renderer subsequently evaluates, see below.

Content-type handlers for Surft! are supplied by the HTML handler and applet handler modules, described below. The other handlers module supplies

handlers for plain text documents, graphical image media and "Helper Applications" as defined by the user's .mailcap file.

The HTML Handler

Perhaps the most important content-type handler, and by far the largest component, in Surft! is the HTML document parser and renderer. It is based upon Stephen Uhler's `html_library` 0.3 public release. In fact, to avoid versioning problems when new versions of the library were released the protocol module was designed around the interfaces to the `html_library` parser and rendering procedures.

`html_library` had to be modified to work with Surft! to support new features included in the browser. The HTML parser could not handle incrementally loading a document. There were particular problems when a tag was split across two network packets. The parser was changed to detect when this occurred, and to push data back into the input stream to the point where all tags in the data fragment were complete. Also, the parser filtered the HTML data into a single Tcl string which it then evaluated to render the document. A mechanism was provided to set a flag which would cause rendering to stop. However, this mechanism did not work reliably, and there was no means by which the application could cache the parsed document. To solve these problems the parser was modified so that it split the resultant Tcl commands into groups (lists within a list) and returned the parsed document back to the application, which would then evaluate it. At this point the application could cache the parsed data. Surft! then evaluated each group of commands in an idle handler to render the document. Performance was improved by populating a Tcl array with each group of commands indexed by a number, thus eliminating excessive list handling in the idle handler.

It is very easy to add handling for new HTML tags to the `html_library`. For example, Uhler's `html_library` distribution includes a new `<COLOR>` tag which is implemented in only 11 lines of Tcl code. Perhaps it is too easy since the temptation to extend HTML in a non-standard fashion becomes almost irresistible! Surft! has added handling of tags for proposed new HTML standards - elements for HTML v3.2

[Raggett96c] and tables [Raggett96a]. An example of the display of a table is shown in figure 3.

Below is a simple table. Nothing fancy.


	Column 1	Column 2
Row 1	Data 1,1	Data 1,2
Row 2	Data 2,1	
Row 3	Data 3,1 with an awful lot of text in it and a nested anchor	<ul style="list-style-type: none">• List item #1• List item #2

Figure 3 A Table Rendered by SurfIt!

Tables are implemented by creating a new Tk Text widget for each table cell and using the grid geometry manager to layout the widgets in a two-dimensional grid. This made it necessary to add the ability to nest Text widgets to the `html_library`. Adding this functionality proved to be a non-trivial task, since some parsing attributes had to be carried through to the nested widget, such as font settings, but some attributes were specific to the nested widget, such as indentation and word wrapping. `html_library` used a single Tcl array variable to contain state information for the parsing and rendering process. It used the widget pathname of the hyperwindow's Text widget into which the document was being rendered to form this variable name, and the pathname was passed to the rendering procedures. Unfortunately, the pathname was fixed at the time the document parsing started, making it inconvenient to nest widgets.

The new table-enhanced version of `html_library` splits the state information into two places. Information that pertains to the parsing process and affects all rendering is held in a single array variable whose name is formed from the hyperwindow's Text widget pathname. This variable also contains the stack of nested widgets, so that the currently active widget is easily accessible. Window specific rendering information is held in array variables whose name is formed from the nested widget's pathname.

This approach has been found to be robust when displaying tables and also supports nested tables. This facility will now make it possible to implement a compound document architecture proposed for

HTML - the OBJECT element [Raggett96b]. The only problem now with tables is that the Tk Text widget itself needs to be able to automatically resize itself to its contents, and it needs to be able to scroll its contents on a pixel basis, rather than on a line basis, since embedded windows (which are used to display graphic images and tables) appear to the Text widget to be part of a single line and so have undesirable scrolling behaviour.

SurfIt! includes a system for sizing a Text widget: a kludge using the Text widget's scroll commands. To achieve pixel-level scrolling this workaround is used to embed the Text widget in a Canvas widget. The Canvas widget is then actually scrolled by the scrollbars. Scrolling using this method is very slow and cumbersome.

Performance

The Tcl HTML renderer is very slow to display a document due to its highly iterative nature. It is interesting to note that the *parser* is not the cause of slow document display, even though it makes several passes over the document text. Profiling reveals that the critical section of code in the renderer is the `HMrender` routine which is invoked for every HTML tag in the document. Work is continuing to improve the performance of the rendering engine in the Tcl code. The `HMrender` procedure has been implemented in C code which makes a substantial performance improvement.

Active Message Content

The applet module provides a content-type handler for application/x-tcl documents. These "documents" are executed rather than visually displayed and allow the implementation of active message content. Such small programs are commonly known as applets. Applets may manipulate a hyperwindow, as in figure 2, or they present their own separate user interface, as in figure 4, or they may do both.

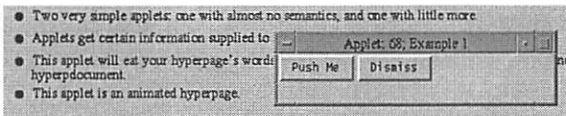


Figure 4: An Applet's Toplevel Window

To execute an applet a safe slave interpreter is created and the applet script is evaluated in that interpreter. A safe slave interpreter has two main advantages: the applet's state and namespace is completely isolated and the applet is prevented from accessing system functions which could result in harm to the user's computing environment. However, an applet which executes in such an environment can provide no useful functions if it has no way of interacting with the user and the browser. Such interaction must occur only in a controlled fashion. Here, a trade-off occurs. The less restriction that is imposed on the applet the more useful functions it can perform, but the risk of damage to the user's computer system increases. SurfIt! seeks to place a level of restriction upon an applet such that the applet can cause no permanent harm.

Security, where the aim of the system is to prevent the state of the computer running the applet from being modified by the applet in an undesirable way, is not the only issue to be dealt with when offering an applet technology. In addition, the applet must be prevented from consuming an unfair amount of system resources, including CPU and memory usage, and network bandwidth. Some resources, such as memory, may need to have some absolute limit set, but for other resources, such as CPU or network utilisation, it is not the total consumption of resources that needs to be limited, but rather the rate at which those resources are consumed. In this case there are no good heuristics available for determining whether an applet has exceeded its fair share. Instead, SurfIt! provides instrumentation which allows the user to monitor resource consumption, and it provides the means to suspend or terminate an applet.

SurfIt! must also concern itself with safeguarding the user's privacy. The overriding policy of SurfIt! is to not make information private to the user available to an applet, since the applet might then transmit that information to a host on the Internet. However, in some circumstances an applet may gather information

about a user using information supplied by the browser. An example of this would be where an applet used anchor activation meta-events (see below) to track which Web pages the user has visited. This may be useful marketing information, and the applet is able to transmit it to an Internet host without the user's permission. Preventing this functionality would unduly prohibit other useful, innocuous applets. To reduce the impact of activities such as that described above SurfIt! allows an applet to only gather information about one hyperwindow. When an applet is loaded into a hyperwindow the user should consider that hyperwindow compromised and untrustworthy. It is then the user's choice to start a new, fresh hyperwindow if desired. Also, at any time the user may terminate applets running in a hyperwindow, which will return the hyperwindow to an uncompromised state.

As mentioned above, SurfIt! has several objects for managing browser functions - hyperdocuments, hyperwindows and the browser itself. These objects form a hierarchy in that order, with the browser object being the highest. All applets are attached to a particular object within the browser, the default being the hyperdocument from which they were loaded.. This attachment defines at which level the applet is operating. Applets may change their level, but may only ascend the level hierarchy. Applets are allowed access to certain browser functions and objects depending on their level. So if, for example, an applet wishes to become independent of the hyperwindow to which it was originally attached it must attach itself to the browser, but then it loses access to all hyperwindow objects, including the hyperwindow to which it was previously attached. This ensures the privacy of all hyperwindows.

Certain semantics are defined at attachment levels. For example when a new hyperdocument is loaded all applets attached to the previous hyperdocument are terminated. Similarly, when a hyperwindow is destroyed all applets attached to the hyperwindow (and all child objects) are terminated.

Applets may be associated with HTML fill-out forms (using the SCRIPT attribute of the FORM tag). This forms the fourth and lowest level of attachment.

Applets at the form level cannot change level, and cannot attach to other forms within the same hyperdocument. An applet attached to a form has access to all of the form's input items.

Tcl Applet API

To be able to perform interesting and useful functions some applets will need to obtain information from the browser. They will also need to be informed of various "meta-events" that occur during the execution of the browser, such as anchor activation, hyperdocument loads and document display completion, and so on. Surflt! defines an application programming interface (API) which applets may use to interact with the browser.

The Tcl Applet API is currently under development and will change in the future, but it currently has two methods for applet interaction: an 'applet' command and by means of application "call-ins".

The applet command provides several methods to control aspects of the browser/applet interaction. These methods include commands to query the browser type, return the hyperwindow which loaded the applet and where it was loaded, change the applet level as well as commands to load a new document into the hyperwindow or fetch data given a URL.

Application call-ins are used to inform the applet of a meta-event. A call-in is a procedure prototype that is invoked when that meta-event occurs. The currently defined call-ins include procedures to notify that the applet is to be terminated, that a new form item has been created, that the user has activated an anchor, that the hyperdocument has finished loading, and so on.

Call-ins pollute the applet's procedure namespace and have the potential to cause programming errors. For this reason the call-in mechanism will be replaced in a future version of Surflt! by a callback mechanism in the applet command.

The Safe-Tk Module

The only means by which an applet can interact with the user is via Tk. However, Tk has not been designed with safety in mind. There are many potential security threats which must be dealt with before an applet should be allowed to create and

modify Tk widgets. Also, Tk does not currently handle the multiple interpreters now available in Tcl version 7.5.

Analagous to Safe-Tcl is Safe-Tk [Ball96b]. Safe-Tk is a redesign of the Tk widget set that supports multiple interpreters and that also supports the concept of safe access to Tk. Surflt! includes a prototype implementation of the Safe-Tk design.

Safe-Tk continues to support only a single widget heirarchy, but different views of that heirarchy can be created by the widget path equivalent of "chroot". Once a safe interpreter has been granted access to a widget subheirarchy all widgets in that subheirarchy are considered to be compromised and the untrusted script may manipulate them as it sees fit. Safe-Tk allows this but ensures that the user is aware of which widgets are compromised (by some osrt of visual indication) and that widgets that have not had access granted to them are not also compromised. Surflt!'s applet module grants an applet access to the hyperdocument which loaded the applet and also automatically creates a toplevel widget which is aliased to the "." widget in the applet's interpreter.

The Safe-Tk system creates command aliases in a slave interpreter for the widget that it has been granted access to, and for all widgets that are descendents of that widget. Aliases are also created for the various widget class commands. Whenever the slave interpreter configures a widget to perform an action, the Safe-Tk system ensures that the action executes in the slave interpreter. Safe-Tk allows many separate interpreters to each define a widget action and for those actions to all be executed in their respective interpreters, but the current Surflt! prototype does not yet handle this case.

Safe-Tk also defines command aliases for other Tk commands. Trusted scripts (ie. scripts running in unsafe interpreters) are granted full access to these commands, but restrictions are placed on these commands when they are defined in safe interpreters and some commands are not defined at all.

The most obvious security threat in Tk is the send command. This command is not defined in a safe slave interpreter. Certain other Tk commands cause less obvious security problems. For example, the

applet could mount denial-of-service attacks: the grab command would allow an applet to freeze up the entire display with a global grab and with the selection command the applet could continually clear the selection or perhaps set it to some obtrusive message. The various other Tk commands all have particular potential undesirable effects checked and disallowed.

Hypertools

The main focus of SurfIt! is to display World Wide Web documents, ie. it is a WWW user agent (a.k.a. browser). SurfIt! is not a mail user agent, nor is it a USENET news user agent. SurfIt! does not support continuous media (nor is it likely to since Web network protocols are unsuitable for time-critical media). What SurfIt! needs is to be able to call upon some other application to handle those functions when they are encountered, but at the same time to be able to have fine-grained interaction with those tools. This would result in a tight integration of the two applications.

Tk has long held the promise of allowing applications to focus on their main task and communicate with other tools to implement related functions, similar to the way in which small Unix tools can pipe their results to solve complex tasks. This is the concept of "hypertools", made possible by the Tk send command. Hypertools allows applications to remain loosely coupled (ie. they are independent, stand-alone applications), while also allowing for their tight integration. SurfIt! actively supports the hypertool concept.

For all of the tools involved to remain focussed, there will be different hypertools used for different functions. From the perspective of a Web browser there may be hypertools for handling various functions: email, USENET news, and so on, as well as hypertools for display continuous media. For this reason SurfIt! defines classes of functions. However, the good thing about hypertools, like standards, is that there are so many to choose from. For each function class there may be several applications

available for use as a hypertool. For example, exmh and TkMail can both handle email.

SurfIt!'s hypertool module provides a registration mechanism to associate an application with a function class. The user may then register her favourite application for a particular class. However, there are further problems. For a given function class SurfIt! needs to know which Tcl commands to send to the hypertool to implement the functions of that class. However, different applications will implement their functions using different procedures and arguments. For example, to compose a new email message (perhaps in response to the user activating a mailto: URL), exmh uses Msg_Compose and TkMail uses mfv:compose. It is essential for SurfIt! to remain independent of these details, so some sort of hypertool protocol must be created for each hypertool class. The hypertool module includes an (incomplete) prototype protocol for continuous media.

Future Developments

There are several improvements planned for SurfIt!. First and foremost is to bring SurfIt! out of alpha release and into beta release. This will mainly involve expanding and stabilising the prototype Tcl applet APIs. Also, general browser functions such as hotlists will be implemented.

The World Wide Web is in a state of rapid and constant evolution of standards. SurfIt! will implement new standards and track their development.

Style Sheets

A major improvement planned for SurfIt! is the implementation of style sheets [Lie]. Style sheets allow the separation of a documents content and structure from presentational information.

The HTML module is well placed to implement style sheets, since it is already well parameterised. However, work will need to be done to handle cascading style sheets and handling nested Text widgets.

Compound Documents

SurfIt! currently relies on the HTML anchor element to embed Tcl applets in a Web page. However, this element is deficient in various ways for this purpose, particularly with respect to nested anchor elements. The proposed INSERT element for a compound document architecture not only improves the method for embedding applets but also provides a generalised scheme for embedding arbitrary media types in a document.

Java

The Java language and associated run-time environment have made a tremendous impact on the World Wide Web. Tcl is in no way incompatible with Java, and the two languages can add value to each other. There are already efforts underway to bring a Tcl/Tk capability to Netscape/Java [Levy]. SurfIt! will complement these efforts by introducing the opposite feature by allowing Java classes to extend a Tcl applet's interpreter. Tcl applet developers will then have a similar facility to that which Tcl application developers have now where C or C++ code may be used to extend the functions of a Tcl interpreter. The Tcl/Java interface [Stanton] will be used as the initial basis for this work.

Conclusion

SurfIt! is a general-purpose World Wide Web user agent that has been implemented entirely as Tcl/Tk script code. It has most of the features of modern Web browsers, including inline images, other HTML v3.2 elements, concurrent document download with incremental display, local caching, and so on. SurfIt! currently lacks certain critical features, such as hostlists and printing, to make it valid choice for casual users, thus restricting its use to research purposes. However, it is planned to add all of the necessary features for general-purpose use in the near future.

SurfIt! supports active message content using applets written in the Tcl/Tk language. These applets are executed in a separate slave interpreter and the host computer is protected from any undesirable actions of the applet. A new API has been created to support

the functioning of Tcl applets. An extension to Tk, Safe-Tk, has been designed and prototyped which allows multiple interpreters in a Tk application, including safe interpreters. Although part of the original motivation for developing SurfIt! was the lack of essential functionality in Java's AWT, which is now not very apparent, this work has been continued because of Tcl's benefits in improving programmer productivity, with a view to later integrating the Java/AWT run-time system into SurfIt!.

Hypertools are an essential part of SurfIt!'s multi-protocol, multi-media strategy. Using the hypertool concept, applications such as SurfIt! can focus on their primary task and use standard protocols, based on Tk's send command, to integrate with other hyper-applications.

SurfIt! is being developed and extended in step with developments in the World Wide Web. Most of the major new standards are to be implemented

Acknowledgements

The author wishes to thank Jacob Levy and Stephen Uhler of Sun Microsystems Laboratories for their help in making SurfIt! possible.

I would also like to thank all of the SurfIt! alpha-testers & contributors, in particular Peter Farmer and Tom Tromeey.

References

- [Ball95] Ball, S. *New Approaches to Custom WWW Interfaces*, Proceedings of the 1995 AUUG/APWWW Conference, Darling Harbour, Sydney Australia, September 1995.
- [Ball96] Ball, S. *SurfIt!* World Wide Web browser software.

<http://pastime.anu.edu.au/SurfIt/>
<ftp://pastime.anu.edu.au/pub/SurfIt/surfit.tar.gz>

- [Ball96b] Ball, S. Safe-Tk Design Document.
<http://pastime.anu.edu.au/steve/safe-tk-design.html>
- [2] *tkWWW* World Wide Web browser
version 0.12.
<http://www.mit.edu:8001/afs/athena.mit.edu/course/other/cdsdev/html/welcome.html>
- [Raggett96a] Raggett, D. *HTML3 Table Model*.
<http://www.w3.org/pub/WWW/TR/WD-tables.html>
- [Raggett95b] Ragget, D, et al. *Inserting Objects into HTML*.
<http://www.w3.org/pub/WWW/TR/WD-object.html>
- [Raggett96c] Raggett, D. *Introducing HTML v3.2*.
<http://www.w3.org/pub/WWW/Markup/Wilbur>
- [Lie] Lie, Hakon. Cascading Style Sheets
Draft Specification.
<http://www.w3.org/pub/WWW/Style/css/>
- [Stanton] Scott Stanton and Ken Corey. *The TclJava Demonstration*.
<ftp://ftp.smli.com/pub/tcl/tcljava-0.1.tar.gz>
- [Uhler95] Stephen Uhler, Sun Microsystem
Laboratories.
Personal Communication.
suhler@eng.sun.com
- [Levy95] Jacob Levy, Sun Microsystem
Laboratories.
Personal Communication.
Jacob.Levy@eng.sun.com

Tcl/Tk HTML Tools

Brent Welch

Steve Uhler

{bwelch,suhler}@eng.sun.com

Sun Microsystems Laboratories

2550 Garcia Ave. MS UMTV29-232

Mountain View, CA 94043

Abstract

This paper describes tools and techniques that support HTML processing with Tcl and Tk. The tools include an HTML parser, a table-driven display engine, and WebEdit, which is a WYSIWYG editor for HTML documents. The parser and display engine are written as a small library that can easily be added to applications that need to display HTML documents. The library has a modular implementation so that applications can customize and extend the library for specialized needs. In particular, WebEdit extends the library to provide an authoring environment. The editor uses the tag and mark facilities of the Tk text widget as the primary data structure for the representation of HTML within the edited document. The paper also describes the performance issues associated with the text widget and one optimization to its implementation.

1 Introduction

HTML is becoming a defacto standard for documents because of its support for formatted text, images, and hypertext links. Not only is it used in the global Internet, but closed intra-nets often use HTML for shared documents and corporate knowledge bases. Many applications are being reimplemented to use HTML, especially forms, for their user interface. This can be awkward, however, and requires complex cgi-bin programming to implement the application in the client-server browser context. An alternative approach to using standard browsers is to embed support for HTML into existing applications.

This paper describes a small script library that makes it easy to support HTML for systems that use Tcl and Tk [Ousterhout94][Welch95]. The library is compact and efficient, and its table-driven implementation makes it easily extensible to support new HTML tags. An application can define new tags, or

override the semantics of standard tags, to support direct control of the application from the HTML document.

The fundamental operation of the library is an `Html_Parse` operation that applies a function to each HTML markup tag in a document. Different functions are applied to the HTML to achieve different effects. For example, the `Html_Render` function displays the document in a Tk text widget. The `Html_Validate` function validates URLs in hyperlink and image tags. The library also includes supporting functions to deal with fetching URLs and submitting forms.

Authoring is clearly an important part of an HTML-oriented environment. This paper also describes a WYSIWYG editor for HTML, WebEdit, that is based on the HTML display library. WebEdit supports basic markup, lists, hypertext links, images, imagemaps, and forms. Table support is in progress, and has been demonstrated by other tools [Ball96] that also use our display library. WebEdit is also a browser, so you can roam the web and assemble pages from parts of other pages. This differentiates it from other tools that provide similar features such as Adobe's PageMill™.

The following sections describe our tools in more detail. Particular attention is paid to some interesting Tcl programming techniques that can be applied to a variety of Tcl applications.

- The parser maps data into a Tcl program that is then evaluated to process the data.
- The editor uses the tag and mark facilities of the Tk text widget as its data structure to represent information about the HTML.

2 The HTML Display Library

The HTML display library implements HTML/2.0 in about 1300 lines of Tcl code. This includes support for basic formatting tags, hypertext links,

images, forms, and some simple extensions (e.g., centering, colors, font size). A simple browser requires a couple hundred more lines for a decent user interface. A stripped down version of the library that just handles local files and local images (i.e. no forms or HTTP) takes about 600 lines of Tcl.

The library architecture is extensible to make it easy to add support for new HTML tags*. The library is organized into the building blocks described below:

- The heart of the display library is `Html_Parse` that maps a function onto each HTML tag. The caller of `Html_Parse` specifies a function and possibly some initial parameters, and then `Html_Parse` calls that function with some additional parameters for each HTML tag in the document.
- The `Html_Render` procedure uses the Tk text widget to display HTML. It is invoked as the callback function from `Html_Parse`. `Html_Render` is described in more detail later.
- The `Html_Init` and `Html_Reset` procedures configure a Tk text widget for use with `Html_Render`. `Html_Init` is called once to define various Tk text tags, and `Html_Reset` is called before each new page is displayed.
- `Html_Render` maintains several state stacks that control different aspects of the display such as font, spacing, and margins. Each HTML tag can have an entry in a table, `htmlTagMap`, that define which state stacks it affects. Another table, `htmlBreakMap`, defines which HTML tags cause line breaks.
- Each HTML tag can have an associated helper procedure that is called by `Html_Render` for special processing. The names of these procedures include the tag name so `Html_Render` can call them automatically. Examples include `HtmlTag_img` and `HtmlTag_/form`.
- The helper procedures for hyperlinks, images, and forms have an additional layer so that the library can implement their display while allowing the application to provide the semantics for these elements. The callbacks defined by the library are

*. The term "tag" is used in two contexts within this paper: HTML markup tags that appear in HTML documents (e.g., ``), and Tk text widget tags that are symbolic names for ranges of text within the Tk text widget. The qualified terms "HTML tag" and "Tk text tag" are used to differentiate the two uses of "tag".

`Html_LinkSetup`, `Html_SetImage`, and `Html_SubmitForm`. There are sample implementations of these callbacks that are suitable for a regular web browser, but an application could redefine them for its own purposes.

- The library includes an HTTP package that uses the Tcl7.5 socket facilities to fetch URLs.

3 Implementation of the Display Library

The library uses two basic techniques to get a small and efficient implementation: table-driven programming and dynamic code generation. The table-driven techniques are accomplished by defining Tcl arrays that are indexed by the name of the HTML tag, and by using procedure names that are derived from the name of the HTML tag. The dynamic code generation involves transforming input data into Tcl programs and then using the `subst` or `eval` commands to process the result.

3.1 HTML Parsing

The `Html_Parse` procedure rewrites its HTML input into a Tcl program. The basic idea is that `regsub` is used to look for HTML tags that are delimited by `<` and `>`, and these are converted into Tcl procedure calls. For example:

```
<tag param=value>some text</tag>more
text.
```

gets rewritten into the following Tcl script:

```
Html_Render $win hmstart {} {} {}
Html_Render $win tag {} {param=value}
{some text}
Html_Render $win tag {/} {} {more text}
Html_Render $win hmstart {/} {} {}
```

The caller of `Html_Parse` specifies the name for the procedure in the generated code and optionally some parameters (e.g. `Html_Render $win`). The additional arguments to the procedure are:

- tag, an HTML tag.
- not, either `"` or `/`.
- param, the parameters from the HTML tag.
- text, the text up to the next HTML tag.

The first and last commands of the generated code make the callback with a pseudo-tag (e.g., `hmstart`), as if there were an extra `<hmstart>` and `</hmstart>` around the whole piece of HTML. As described later, application-specific initialization can be associated with the pseudo-tag. A call to `Html_Parse` looks like this:

```
Html_Parse $html \
```

```
[list Html_Render $tkwin] hmstart
```

The basic strategy of `Html_Parse` is to first protect any Tcl special characters that are in the input data. It is important that these do not interfere with the `eval` done later. Next, a regular expression substitution re-writes the input data into a series of Tcl commands. The input data is passed as arguments to the commands. Finally, `eval` is used to run the dynamically generated code. Here is its implementation.

```
proc Html_Parse {html \
    {cmd HhtmlTestParse} \
    {start hmstart}} {
    # Convert Tcl specials to HTML entities
    regsub -all \{ $html {\&ob;} html
    regsub -all \} $html {\&cb;} html
    regsub -all \\\} $html {\&bsl;} html
    set w " \t\r\n";# white space
    # Expression to match HTML tags
    set exp <(/?)(\^[^$w]+)\[ $w]*(\^[^>]*)>
    # Re-write pattern:
    # \1 is either / or the empty string
    # \2 is the HTML tag
    # \3 is the parameters to the tag
    set sub "\}\n$cmd {\2} {\1} {\3} {\{"
    regsub -all $exp $html $sub html
    eval "$cmd {$start} {} {} \{$html\}"
    eval "$cmd {$start} / {} {}"
}
```

There are five passes through the HTML input: four global `regsub`s and one `eval`. The first three `regsub` commands replace all curly braces and backslashes with entities (i.e., `&ob;`, `&cb;`, and `&bsl;`;) so they do not interfere with `eval`. Backslashing these characters won't help because things are grouped with braces in the generated code. The entity encoding is used because `Html_Render` already has to decode entities for HTML special characters like `'>'` and `'<'`. The decoder for these entities is described shortly. The fourth `regsub` command picks out the HTML tags and their parameters, and does the rewriting.

Note that each rewrite begins with a close brace and ends with an open brace. This groups the unmatched text between the HTML tags. The first `eval` command supplies the balancing braces. Profiling measurements indicate that page display time is dominated by the Tcl parser and the Tk text widget, not by the regular expression substitutions.

There are two bugs in this version of the parser. The first is that HTML comments are not handled reli-

ably because they can contain `'>'` characters. Similarly, the parameter values in HTML tags could also contain `'>'` characters. Comments can be correctly handled with another pass that maps their syntax into something compatible with the matching done last. For display the comments can be completely deleted. For editing, we divert the values of the comments into an array, and replace the comments with HTML tags that reference the array.

For maximum speed and reliability we will soon rewrite the `Html_Parse` procedure in C. It will remain relatively simple, however, and retain the basic strategy of calling a Tcl procedure to handle each HTML tag. The ability to map different functions over the HTML has proven to be quite useful in the editor and other HTML document management tools.

3.2 An Overview of `Html_Render`

The `Html_Render` procedure displays HTML in a Tk text widget. It is called from `Html_Parse` for each HTML tag in the document. It maintains a state machine to determine how text is formatted. The state machine is table-driven, and any special cases are handled by special per-tag helper routines.

The basic steps to `Html_Render` are shown below. The parameters (e.g., `tag`, `not`, `param` and `text`) were described in the previous section. The following subsections describe some of these steps in more detail.

- Decode any entities in `text`.
- Push or pop state associated with `tag`.
- Decide if a line break should occur, and deal with white space.
- Call a per-tag helper procedure, `HtmlTag_ $not $tag`, for special processing, if any.
- Compute a set of Tk text tags to apply to `text`.
- Insert `text` into `win` with the current set of Tk text tags.

3.3 Decoding HTML entities

Decoding HTML entities provides another example of dynamically generating Tcl code to process data.

An entity encodes characters like `'<'` and `'>'` that are special to HTML. If they are to appear in the document as literal `'<'` and `'>'` characters, they need to be encoded so they are not interpreted as HTML

markup. Characters that have their high-order bit set (e.g., '©' and 'â') are also encoded. The encodings are keywords or decimal values that are enclosed with '&' and ';' , like these:

```
Copyright &169; less than &lt; greater
than &gt;
```

The basic idea of the decoder is that it first replaces entities with a `format` command that will generate the real character. The `subst` command is then used to replace the `format` commands with the special character. Here is the code for the entity decoder:

```
proc HtmlDecodeEntity {text} {
    if {[regexp & $text]} {return $text}
    regsub -all {[[] [$\]]} $text \
        {\|\|1} new
    regsub -all {&#([0-9][0-9]?[0-9]?);?} \
        $new {[format %c \
            [scan \1 %d tmp;set tmp]]} new
    regsub -all {&([a-zA-Z]+);?} $new \
        {[HtmlMapEntity \1]} new
    return [subst $new]
}
```

The first `regexp` just checks to see if any work really needs to be done. The next `regsub` is a pre-pass to quote all the Tcl special characters. This is necessary so that `subst` doesn't interpret the wrong things. The next `regsub` command replaces the decimal-valued entities with a `format` command. The `format` command uses `scan` to interpret the decimal values to avoid cases like "&09;" that are otherwise incorrectly interpreted as invalid octal numbers by `format`.*

The named entities require a table that maps from the entity name to a character code value. Access to the table is done by the procedure `HtmlMapEntity` to make error handling easier. A subset of `htmlEntityMap` is shown below.

```
array set htmlEntityMap {
    lt < gt > amp & quot \" copy \xa9
    ob \x7b cb \x7d bsl \|
}
proc HtmlMapEntity {text {unknown ?}} {
    global htmlEntityMap
    set result $unknown
    catch {
        set result $htmlEntityMap($text)
    }
}
```

*. If the HTML specification for entities used hexadecimal, the `format` command could be written simply like this:

```
format %c 0x\1
```

```
return $result
}
```

Note the difference between using `eval` in `Html_Parse` and using `subst` in `HtmlDecodeEntity`. `Html_Parse` has to handle every character, even those not matched by the `regsub` pattern. The clever placement of curly braces groups *unmatched* text into a command argument. `Eval` is necessary to pass that argument to a procedure. In `HtmlDecodeEntity`, only the matched text has to be processed, and the unmatched text should not be modified. The `subst` only affects text matched by `regsub`. It's also easier to quote Tcl special characters because `subst` is less sensitive to curly braces.

3.4 Table-Driven Display State

One of the more interesting table-driven techniques concerns the display state. There are several orthogonal properties that combine to affect the way formatted text is displayed. These properties include the margins, line spacing, and the font. The font itself is determined by several properties including the font family (i.e. typeface), size, style (e.g. bold, italic, underline), and color. `Html_Render` keeps a separate state stack for each of these individual properties, and the `htmlTagMap` table defines how a given HTML tag affects the stacks. A subset of `htmlTagMap` is shown below:

```
array set htmlTagMap {
    b {weight bold}
    code {size 12 family courier}
    em {style i}
    h1 {size 24 weight bold
        Tspace hspacebig}
    h3 {size 16 weight bold
        Tspace hspacemid}
    ol {indent 1}
    u {Tunderline underline}
    pre {fill 0 family courier size 12
        Tnowrap nowrap}
}
```

The key to the map is the name of the HTML tag (e.g. `h1`). The value is a list of name-value pairs. The name identifies a stack, and the value is pushed onto the stack when the HTML tag appears. When the corresponding close tag appears (e.g., `/h1`), the values are popped. The effects of most of the HTML tags can be completely defined by their entries in the `htmlTagMap`, which means there is no need for a per-tag helper procedure.

The `HtmlStack` and `HtmlStack/` procedures push and pop values from the stacks, respectively. The pop routine has a funny name, `HtmlStack/`, and the same arguments as the push routine, `HtmlStack`. This lets `Html_Render` do the pushing and popping for the current tag with a single Tcl command shown below. The `catch` is necessary because there may not be an `htmlTagMap` entry for all tags.

```
catch {
    HtmlStack$not $win $htmlTagMap($tag)
}
```

3.5 The Helper Procedures

Some HTML tags require special processing. This includes tags in the HTML header (e.g., `<title>`), list-related tags (e.g., `` and ``), link tags, image tags, and form-related tags. The special processing is implemented by Tcl procedures that include the name of the HTML tag in their name. This makes it easy to extend the library to support new tags.

`Html_Render` attempts to call a per-tag handler as shown below. Again, `catch` is used because there may not be a tag handler. The `text` parameter is passed by name, not by value, so the tag handler can side-effect the text before `Html_Render` inserts it into the text widget.

```
catch {
    HtmlTag_$not$tag $win $tag $param text
}
```

The tag handler is illustrated with an example that defines a `color` tag. This is a nonstandard HTML tag that lets you specify the color for text. It takes one parameter that specifies the color. Specifying red text would look like this:

```
<color value=red>This is red.</color>
This is not.
```

The job of the `HtmlTag_color` procedure is to get the value and manipulate the state stacks so that the following text is red. The job of the `HtmlTag_/color` procedure is to undo the effect by popping the stack. A new stack named `Tcolor` is introduced to support this. Here is the code:

```
proc HtmlTag_color {win param textVar} {
    set value bad_color
    HtmlExtractParam $param value
    HtmlStack $win "Tcolor $value"
    $win tag configure $value \
        -foreground $value
}
```

```
proc HMTag_/color {win param textVar} {
    HtmlStack/ $win "Tcolor {}"
}
```

The `HtmlExtractParam` procedure picks out values from the `Name=Value` syntax used in HTML tags. This uses regular expressions, too. While it is not possible to parse all the parameters at once with regular expressions, it is possible to pick out a single parameter at a time.

The second argument to `HtmlExtractParam`, in this case `value`, specifies the name of the parameter. If it exists in the parameter list, a Tcl variable by that name is initialized to the value (e.g. `red`). The variable is not defined if the parameter isn't specified in the HTML tag. In this case the color would remain `bad_value` and the tag `configure` would fail. `Html_Render` ignores errors from tag helper procedures, except during debugging.

3.6 Formatting the Text

The state stacks cause different Tk text tags to be applied to the text. In the simplest case, the value on the top of the stack is used as the name of a Tk text tag to apply to the text. This is done with any stack that is named with a leading `T`. The `Tcolor` stack used for the `color` HTML tag is an example. The values on the `Tcolor` stack are color names that have been configured as Tk text tags that have that foreground color. As another example, the `h1` tag pushes the value `hspacebig` onto the `Tspace` stack. The following initialization code in `Html_Init` configures the `hspacebig` tag to have certain interline spacing:

```
$win tag configure hspacebig \
    -spacing1 10p -spacing3 6p
```

This tag configuration and the entry in `htmlTagMap` are all that is needed to support the `h1` HTML tag.

Other uses of the state stacks are a little more complicated. The current indentation level is determined by the size of the `indent` stack, for example, not its top value. The indent level selects from a set of Tk text tags that are configured to have different indents and tab stops. The font is determined by the combination of the top-of-stack values from the `weight`, `family`, `size`, and `style` stacks.

3.7 Managing Instance Data

The library must keep state information for each text widget that is being used to display HTML.

(The browser and editor display more than one page at a time.) The name of the text widget is passed into the library, which uses `upvar` to map that into the name of a Tcl array. Procedures contain this statement:

```
upvar #0 HM$win var
```

The rest of the code references `var`. For example, all the simple state stacks are found automatically in this `foreach` loop:

```
foreach stack [array names var T*] {  
    # Look at the top-of-stack value  
    set top [lindex $var($stack) end]  
}
```

4 WebEdit, a WYSIWYG HTML Editor

WebEdit* provides a WYSIWYG editing environment where the user is shielded from direct manipulation of the HTML tags. Instead, the user performs logical operations such as making text bold or changing the paragraph type to a heading, and the editor manages the HTML tags. The page is continuously displayed in the format it would be viewed in a browser. It is possible to view the underlying HTML tags from within the editor, but even in this mode the user is prevented from manipulating the HTML tags directly. This may constrain some HTML wizards from achieving bizarre effects, but it also ensures that the page contains valid HTML.

4.1 Cut, Paste, and URLs

The editor is a client of the display library. Cut and paste is done by generating HTML during copy or cut, and rendering HTML during paste. The display library handles paste, and the editor has an output module for cut, copy, and saving HTML to a file. When a range of text is copied or cut, its HTML representation is computed. For example, a selection containing **bold** text would be returned as the following string:

```
this is <b>bold</b>
```

Pasting is just a matter of using the display library to render the HTML markup. This allows cut and paste between pages. It also allows interoperability between plain text tools such as text editors and email readers. If a user selects HTML markup in their text editor, when they paste that into WebEdit it is automatically rendered as formatted text.

*. This name will change because there is an existing product with this obvious name.

Other edit operations are built on top of cut and paste. For example, when the user selects a range of text and makes it bold, the editor cuts the affected region, wraps it in `` and `` HTML tags, and then uses the display engine to redisplay the result.

The editor is also a browser, so you can easily cruise the net and copy links and images from other pages. Copying a URL creates an interesting problem. Suppose a page contains a hypertext link with a relative URL (images have the same problem):

```
<a href=file.html>
```

When this link is copied into another page, the user expects it to work. This may require resolution of the URL into an absolute reference:

```
<a href=http://somewhere.com/file.html>
```

Or, if the page is on the same server but in a different directory, a new relative name may need to be computed:

```
<a href=../otherdir/file.html>
```

The editor does some extra work during copy and paste to transform URLs in this way. First, when it generates a selection that contains a relative URL, a base tag is also emitted as part of the selection:

```
<base href=http://somewhere.com>  
<a href=file.html>
```

During a paste operation, base tags are used to compute the "best" representation of a relative URL in the destination page. (The base tag is not copied into the destination page.) As shown above, the new URL may be an absolute URL or a different relative URL.

4.2 Representing HTML

The default behavior of the display library throws away the information about HTML markup. It just derives enough information to compute the display. `Html_Render` was modified to call into the editor so it can add additional tags and marks to the text widget to represent HTML tags. For example, the display library will use three Tk text tags to display an `h1` heading; one for the line spacing, one for the font, and one for the indent level. The editor adds another text tag, `H:h1`, to all the text in the heading. By querying the current tags whose names begin with `H:`, the editor can detect what HTML tags are in effect in order to reinitialize the state of the display library. Using Tk text tags and marks to represent HTML tags is a natural design decision for the

editor because the text widget maintains the tags and marks as the user inserts and deletes text.

It is tempting to try and combine the tags that represent the HTML markup with tags that affect the display of the text. Unfortunately, the effects of combining HTML tags make reverse engineering the HTML from the display tags difficult. The `strong` tag, for example, implies a different font when applied within a regular paragraph than when it is applied within a heading. There are also different HTML tags that produce the same visual effect (e.g., `b` and `strong`, or `em` and `var`).

Not all HTML tags can be represented by Tk text tags. HTML tags like `img`, `li`, and `input` do not ordinarily have matching close markers (e.g., there is never a `/img` tag.) WebEdit classifies these as singletons and uses text marks to represent them. One minor issue with marks is that they don't get deleted when the surrounding text is deleted. WebEdit must find and delete the marks explicitly.

The editor uses a set of tables to classify HTML tags into classes. It differentiates between styles (e.g., `strong`), paragraphs (e.g., `h1`), lists (e.g., `ol`), list items (e.g., `dt`), structure elements (e.g., `form`), and singletons (e.g., `img`). These classifications affect editing operations. For example, only one paragraph type can be in effect at once, while multiple style types can be in force.

Each HTML tag also has a set of parameters that are valid for it. These are also defined in a table, and the editor has a general property sheet dialog that is used to set these parameters. Currently you have to edit the WebEdit source to update the tables. We plan to add a user interface to add new tags and add new parameters to tags so that WebEdit can be used to author HTML for custom applications.

The tag classifications are also used during output. The main trick to output is sorting HTML tags that occur at the same text index. For example, a heading that is also a hypertext link will have an `<h1>` and `` HTML tag that both start at the same position in the Tk text widget. During output, the editor must decide which HTML tag comes first. The tag classifications are used to define a sorting order among classes, and tags within the same class just sort alphabetically. The sorting order is listed below from highest (i.e., earliest in output) to lowest:

```
close-style
close-paragraph
close-list
close-structure
open-structure
open-list
list-item
open-paragraph
open-style
singleton
```

The HTML produced for the hypertext link in a heading would be:

```
<h1><a href=...>The text</a></h1>
```

The file output routine uses another table to define how to format tags so the HTML file is legible.

4.3 New Features for the Text Widget

The editor's output module that is layered on a new dump operation in the Tk text widget. This is used when saving to a file or when generating a selection. The dump operation either returns information about text widget segments, or it can call a Tcl command with information about each segment. The segments include text, tag transitions (i.e., tagon and tagoff), marks, and embedded windows. The information identifies the type of the segment, its value, and its index within the widget. Each text segment represents a range of text that is not split by a mark, tag transition, embedded window, or newline. The segments are a direct reflection of the data structure used internally by the text widget.

The output module uses the callback form of the dump operation. It looks for the tag transitions and text marks that represent HTML. These are accumulated until a text segment is encountered. The accumulated tags are then sorted and output before the following text.

The editor often needs to know the current range of a tag. That is, given that `tag names` reports a tag in effect at a given index, where does that tag range start and end? This is used when changing a paragraph type and when editing hypertext links. There used to be no practical way to find out the current range; you had to step through every range of the tag with `tag nextrange`. We added the `tag prevrange` operation that is the complement to `tag nextrange`. The two range operations are used together to find the current range. The procedure is shown below. The new `foreach` command is abused to set multiple variables from a command

that returns a list.

```
proc Edit_CurrentRange { win tag mark } {
    foreach {start end} \
        [$win tag prevrange $tag $mark] {}
    if {$end == "" || \
        [$win compare $end < $mark]} {
        foreach {start end} \
            [$win tag nextrange $tag $mark] {}
        if {$start == "" || \
            [$win compare $start] > $mark]} {
            return {}
        }
    }
    return [list $start $end]
}
```

4.4 Performance Issues

We found quadratic behavior in the implementation of text tags that caused a slowdown with pages that had lots of unique tags. The cost of adding a new tag to the text widget was proportional to the number of other unique tags already in the widget, so adding N unique tags in the process of loading a page was $O(N^2)$ in processing time.

Describing the source of the problem requires an explanation of how the text widget represents its contents. The text widget uses a BTree to represent all the lines of the text widget. An example is shown in Figure 1. There are interior nodes, line nodes, and line segments. The BTree keeps the number of children of the interior nodes balanced. This includes the number of lines under each level 0 node. More levels are added to the tree as needed to maintain the balance. Each line has a list of all the segments of that line: marks, tag transitions, embedded windows, and text segments.

Tags are represented by tagon and tagoff segments. These segments are at the leaves of the BTree, but

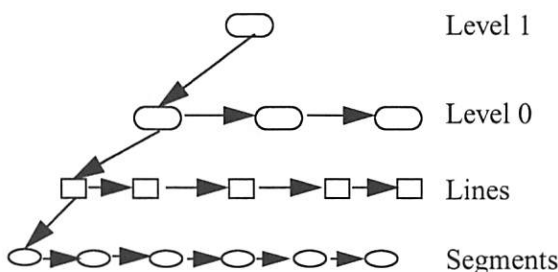


Figure 1. The Text BTree Representation

information about the tag segments is propagated up the tree in the form of summary counts. At each interior node, there is a count of all the tagon and tagoff segments for each tag in all the lines below that node.

By starting at the root it is easy to walk the BTree and only visit nodes that have summary information about a particular tag. This is used for the various tag range operations (i.e., ranges, nextrange, and prevrange).

By starting at a segment and walking up the BTree, you can count up the summary information to determine what tags are active at a particular node. If the count is odd it means the last transition is a tagon and the tag is active.

One problem with this scheme is that there is summary information about every tag in the root node of the BTree. Adding a tag range requires searching the summary list at each node going up the tree from the tag transition segments. The cost of this is proportional to the number of unique tags in the text widget. Therefore adding N unique tags suffers this cost N times, for an overall cost proportional to the square of the number of unique tags. Figure 2 plots the cost of adding and deleting N tags to a text widget. The $O(N^2)$ behavior of the old tag implementation is clear.

We optimized the storage of summary information to prune the tag information out of the interior nodes of the BTree. The information is only necessary below the interior node that covers all the ranges for a given tag. In the best case, a tag that is only in effect on one line has no information at all in the interior nodes. In the worst case, a tag that has an active range at the beginning and end of the widget will have information that does propagate all the way to the root. For applications that have lots of unique tags that are not widely used (like tags that represent URLs in hypertext links), the optimization is very effective. Figure 2 shows that the cost of adding N unique tags that each have a single range is $O(N)$ instead of $O(N^2)$.

The figure shows that the cost of deleting a tag is still proportional to the number of other tags in the text widget. Deleting a tag removes all information about the tag. When this happens, the complete set of tags is searched in order to adjust tag priorities. The priority determines which tag is in effect if

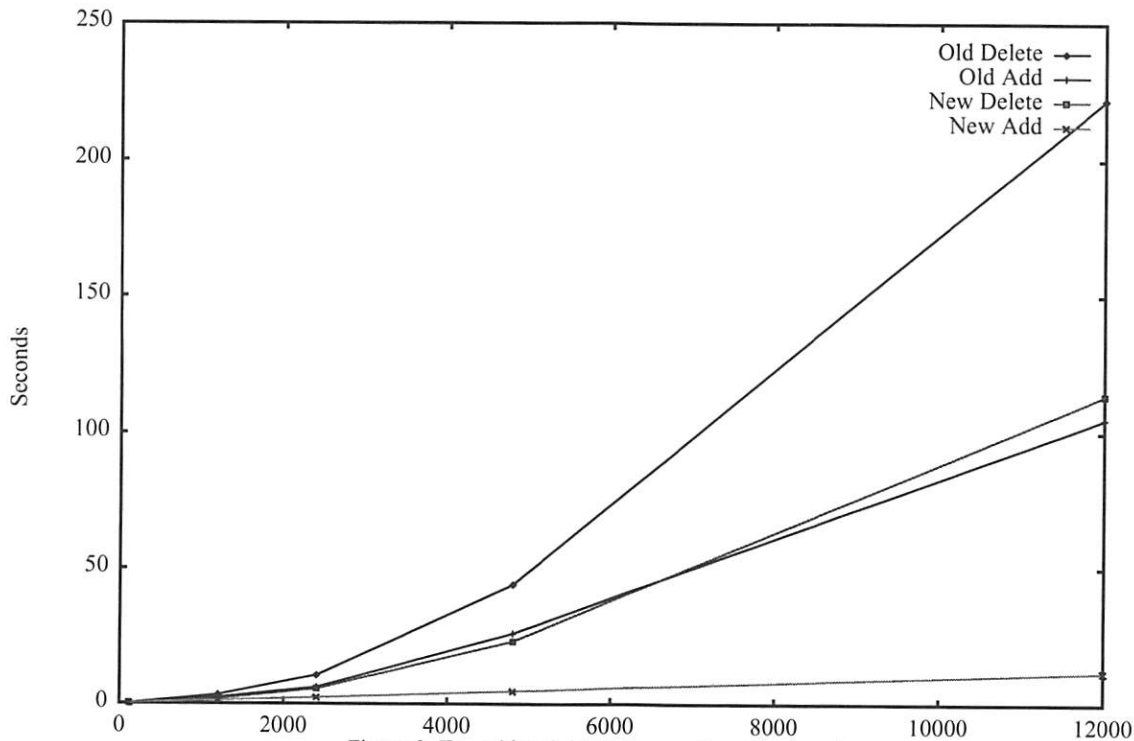


Figure 2. Tag add and delete vs. number of unique tags

more than one tag contributes a display property or a binding.

Note that deleting a tag is different than removing the tag from all ranges of text. The text widget keeps information about a tag even if it has no active ranges. This means that it is good practice to configure your display and binding tags once at the beginning of your application, and then never remove them.

One performance problem remains with the text widget. Complex lines that have lots of segments such as marks and tag transitions are expensive. The BTree does not help to balance this information because each leaf node holds information about one line. A line is defined to end with a newline character, so a long line that wraps many times is still one line. In the limit, if you have no newline characters in your text, it is represented as a linear list of segments, and the BTree is of no value. The tree structure needs to eliminate the distinction between interior nodes and line nodes, and allow the segments for one line to be balanced across nodes.

5 Conclusions and Future Directions

This paper describes a simple and extensible HTML viewer that can easily be added to existing applications. The combination of the generic stack processing and the use of per-tag helper procedures makes it very clean to add support for new HTML tags. In addition, the table-driven programming and dynamic code generation techniques we describe can be used in a variety of Tcl applications.

The WebEdit authoring tool is a natural complement to the HTML display library. The paper describes how WebEdit represents HTML in the Tk text widget, and it describes performance optimizations we have made to the Tk text widget.

The current version of WebEdit is focused on basic HTML generation. This is just a starting point, and we envision a set of related applications that we plan to address in the future:

- Generation of a family of pages from building blocks such as standard headers and images. Any well-managed web site takes this approach in order to provide a consistent look to their pages with a minimum of effort.

- Support for cgi-bin programs that are the standard associated with form processing on the web. WebEdit could create templates and a specification for how the templates get filled with data. The Navisoft servers already let clients embed Tcl in their HTML for this purpose. [Navisoft96]
- Support for client-side functionality in the form of hooks that call functions in the browser or embedding application. This has lots of potential for extending basic HTML with application-specific behaviors. Netscape's Javascript™ [Javascript] is a small step in this direction, but it is currently quite limited. The Netscape browser is closed, so it is not possible to embed it into applications and then use Javascript™ to control the application. In addition, there is no security model for Javascript™, which makes it risky in open environments [LoVerso96]. The use of Safe-Tcl and untrusted interpreters makes it possible to download safe client-side behaviors in open environments.

Availability

The HTML library is available via FTP as:
ftp://ftp.sunlabs.com/pub/tcl/html_library-0.3.tar.gz. The version number may change.

WebEdit will be released shortly. Please visit its page at <http://www.sunlabs.com/people/brent.welch/editor/intro.html> for more information.

References

- Ball96 The SurfIt! web browser.
<http://pastime.anu.edu.au/SurfIt!/>
- Javascript <http://www.c2.org/~andreww/javascript/>
- LoVerso96 <http://www.osf.org/~loverso/javascript/>
- Navisoft96 <http://www.navisoft.com/servdoc/extend/>
- Ousterhout94J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, April 1994, ISBN 0-201-63337-X
- Welch95a B. Welch, *Practical Programming in Tcl and Tk*. Prentice Hall, May 1995. ISBN 0-13-182007-9

Programming the Internet from the Server-Side with Tcl and Audience1™

Adam Sah, Kevin Brown and Eric Brewer

asah@{inktomi.com, cs.berkeley.edu}
kebrown@{inktomi.com, haas.berkeley.edu}
brewer@{inktomi.com, cs.berkeley.edu}

Inktomi Corporation
2168 Shattuck Ave. Suite 210, Berkeley, CA 94704

Abstract

Although innovations on the World Wide Web are currently dominated by exciting client-side products (ie. Java, VRML, Netscape Plug-Ins, etc.), we believe that there is an equally rich server-side programming opportunity. In this paper, we argue that server-side programming will remain an important part of client-server Web applications. We then argue that server language(s) and client-side languages have very different requirements, and don't have to be the same language.

As a running example, we present Audience1™, an end-to-end publishing tool for the World Wide Web, which uses Tcl and MTtcl, a multi-threaded Tcl extension library. Currently, Audience1 is providing web service and mass customization features for the HotBot search engine (a joint venture between Inktomi and HotWired). HotBot can be found at <http://www.hotbot.com>.

I. Introduction

The World Wide Web is a general-purpose, distributed, client-server computing environment. Very high level languages and toolkits play very different roles in each.

On the client side, it is handy to provide tools tuned to the tasks of user interface (UI) design and implementation, security, network communication, installation, configuration, and uninstallation. Thus, the current client-side computing environments all offer a rich set of products to fill these needs, some bundled with the operating system, some as third-party packages and some as both. HTML, Plug-Ins, Java and VRML are rushing to provide these services on the Web.

The server side looks much like the client side, except that user interfaces are less important, and access to sophisticated multi-user data management (ie. DBMSs), report generation, publishing tools and scalability become the dominating issues. Inktomi, BroadVision [Broa95], NeXT [NeXT95], Navisoft [Nav95] and other companies are beginning to develop Web server products that fill various niches of this market. Both Inktomi's and Navisoft's products are based on Tcl.

The remainder of this paper discusses issues in the design and implementation of a sophisticated, scaleable dynamic Web server. In the next section, we review how World Wide Web servers work, why dynamic HTML

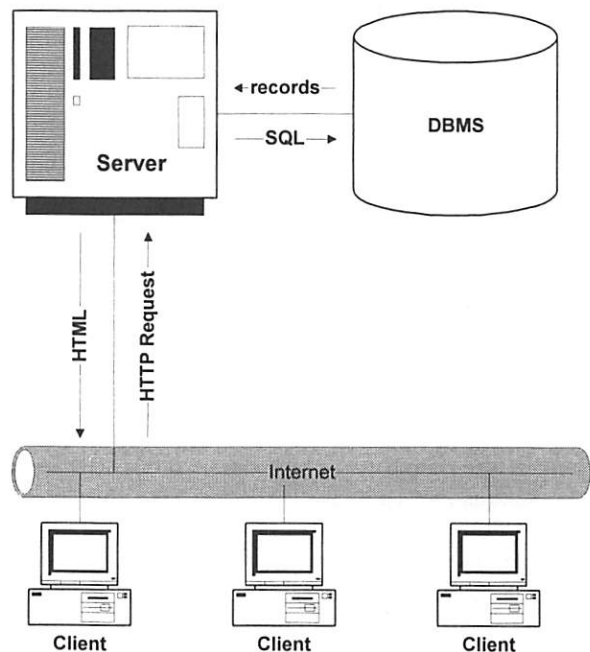


Figure 1. Client-Server computing on the Web

generation is important and why the Common Gateway Interface (CGI) doesn't scale well. In section 3, we explore the value of server-side programming in greater depth and introduce Audience1. In section 4, we

describe the design and implementation issues of Audience1's scripting language, Dynamic Tags™. In section 5, we describe what features of Tcl Dynamic Tags stresses and what backward-compatible suggestions we have for language improvements. Finally, section 6 presents our conclusions.

II. An Overview of Client-Server Computing on the World Wide Web

All Web servers operate over the HyperText Transfer Protocol (HTTP), which dictates how clients and servers communicate. The HyperText Markup Language is one of the data formats supported by HTTP; by convention, all Web browser software (ie. Netscape Mozilla, NCSA Mosaic, Microsoft Internet Explorer) provide HTTP client service and automatic HTML formatting.

First generation Web servers are very similar to file servers: an HTTP request is made on a well-known network port, and a file is sent in reply (data) along with a few simple headers (metadata). In this model, the content delivered to clients is static, which means that all clients are presented the same data, and a given client is given the same data on repeated accesses.

Second generation web servers add the Common Gateway Interface (CGI). Instead of responding to HTTP requests with static content, a program is executed. The output of the program is used as the HTTP response. Since the response is still HTML, but can be different on every request (eg. by including the number of visitors to the site on the page itself), we call this "dynamic HTML generation."

Once in place, there are numerous uses for dynamic content generation:

- content targeting, such as custom newspapers.
- targeted marketing and advertising.
- Web server administration and online publishing tools, customized to the content being edited.
- "do the right thing" presentation: don't offer services that the user can't use or doesn't need. For example, don't present a page containing a Java script for a user not able to execute Java programs.
- "do the better thing" presentation: allow the user access to all services, but make the more useful ones easier to access. For example, in presenting a list of services, put the ones that the individual user is more likely to want earlier in the list.

As a means for producing dynamic content, CGI is flexible, easy to use, and compatible with virtually every programming language. CGI has the downside that it is

slow and scales poorly because it requires a (heavyweight) process to be spawned for each Web request. Popular Web sites can receive hundreds of requests per second (millions per day); examples include Intel's home page (<http://www.intel.com>), stock quote servers, and several search engines.

An alternative to CGI is to bind the language used to create the dynamic content directly into the Web server itself. Although one can use C or C++ for this purpose, the performance of these languages is overkill and their programming models are cumbersome and lead to less robust web services: a scripting language is a better fit.

III. Dynamic Web Service and DBMSs

Dynamic HTML generation is incredibly powerful, but requires some additional features to realize its potential:

user identification. On the Web, users are not only anonymous but are identified only by IP address: it's important to distinguish one user from another so that the Web publisher can tailor or target content to users whose prior behavior or expressed preferences suggest his/her interests. For example, it's useful to ask "what percentage of our users use feature X?"

In Audience1, we include user IDs in the URL and/or a cookie [Nets96]. When a user first visits the site, a unique number is assigned to her, and a version of the site's home page is dynamically generated for her. Later visits automatically present her user ID to the system, although the URL-base scheme requires that she use a bookmark. For example,

<http://www.hotbot.com/UI28736487g> is the URL that user ID "UI28736487g" might use to revisit the site in the future. Bookmarks will include this additional information, and the UID won't interfere with form submissions.

browser targeting. The plethora of Web browsers has left Web publishers in a quandary: if they provide content that all users can see, they cannot use the latest features and have to revert to the least common denominator. If they employ the latest features, they risk alienating the users with legacy browsers.

However, if we know the capabilities of the user's browser, we can produce the best quality content that the browser can handle. For example, JPEG images are generally smaller than GIF images and therefore load faster, but not all browsers can view JPEG images. With browser independence, we can provide the GIF version for less-capable browsers. As another example, many legacy browsers do not support cookies, which

Audience1 uses for user identification; for these browsers, Audience1 automatically embeds the user ID in the URL.

It is easy to discover the browser capabilities because the HTTP request already contains the name of the browser, which can be correlated with a database of browser capabilities. Some capabilities may need to be provided manually by the user, such as size or color depth of the client display.

dynamic HTML rewriting. Many of Audience1's features require specialization of the HTML being sent to the client. For example, Audience1 can rewrite inline image tags to use more advanced image formats if the browser supports them. If the publisher specified foo.gif, but also provided foo.jpg and foo.pjpg, and the client browser supports Progressive JPEG, then foo.pjpg will automatically be specified in the HTML for the page, a rewrite of `` → ``. Rewriting is also used for transmitting user ID information in the URL for hyperlinks pointing to the same site.

persistent storage. Since user-specific data needs to be maintained over a long period of time, and can be difficult to reconstruct if lost, the obvious solution is to store this data in a database management system (DBMS), thus requiring DBMS access to be bound into the Web server as well.

With user identification, client browser identification and DBMS access, we can bring Web services into the mainstream of corporate communication. Unlike paper mail, telephones and faxes other communication media, the Web provides a bi-directional channel, where both channels are computerized and lossless: a company can produce content and receive feedback on that content instantly. Feedback can either be explicit (such as a form the user fills out) or observed (such as which pages are most popular among different user groups). DBMSs provide fast query capability, allowing the company to react to that feedback instantly.

For example:

- If the web server stores logs hits in a DBMS, log analysis can be performed "on the fly". We can target advertising to users based on their behavior.
- If we keep track of a user's preferred language, we can present content in that language, if available.
- We can keep track of a user's user-interface customizations, automatically using these as the defaults during his/her next visit to the site.
- Frequent visitors to the site can be offered coupons and other specials. For commerce sites, frequent buyers can be targeted for special offers.

IV. Dynamic Tags, DBMS Access, and Publishing

Unfortunately, it is not enough to provide user IDs, browser identification, and persistent storage in the abstract. For these facilities to live up to the promises we have outlined we need additional facilities:

database access. Although most DBMS vendors provide pleasant front-end tools for designing, managing, and accessing databases, we need programmatic support from the scripting language. This involves communication issues, parsing issues, data representation issues and memory management issues. These details are beyond the scope of this paper.

Web-based report generation and data access tools. Access must be provided to manage the basic Web data schema (hits, users, pages, etc.). All of these services should be made available through an HTML interface.

schema- and language- extension. It must be straightforward for Web publishers to augment the database to track additional data items. For example, it is common to present a site-specific form to a user when the user creates an account on the site. Since this form contains site-specific fields, the database, report generators and scripting language must all provide facilities to manage this data.

migration from static publishing. It must be possible to migrate an existing site into using the tools of a dynamic Web server. In Audience1, Dynamic Tags™ allow publishers to add small amounts of dynamicism to existing HTML pages, gradually adding more dynamic elements to otherwise-static pages. Unlike the simple dynamic tags that other systems allow, Audience1 tags provide powerful features that hide large, complex add-on services.

Dynamic Tags™ look like HTML tags with an "@" sign prepended to them; unlike normal HTML tags, Dynamic Tags™ are parsed on the server; their output is normal HTML, sent to the browser for further processing. In the context of Java, JavaScript, and other client-side languages, Dynamic Tags™ allow servers to provide the dynamic inputs to the client-side scripts. Java doesn't replace the need for server-side processing because you can't download most databases to clients, and even if you could, there are privacy and security reasons to avoid it. It is also infeasible to transmit large data sets (eg. MRI scans of the user) to the client: users will rarely view more a small portion of large datasets, and so it is more efficient to cull the data set on the server before communicating the result.

One example of Dynamic Tags™ is `<@ad>`, which places an advertisement on the page at the position indicated by the tag. `<@ad-keywords "hard drive">` might find an ad targeted to customers interested in hard drives. Ad targeting is a black art, where good results contribute directly to one's bottom line revenue as more customers "click through" the ad. Therefore, good ad targeting software can consume a large engineering effort spent developing expert heuristics. `<@ad>` and its arguments hide this complexity from the publisher.

The architecture of Audience1 is shown in figure 2. Although pages are served just like NCSA httpd and other traditional web servers, the HTML is intercepted before it is served to the client, and the Dynamic Tags converted to static HTML (executed). Although you have to trust the author of the web server software, we still try to provide a "safe" Tcl interpreter in case a rogue script causes undue harm or is coerced into causing harm by the client's HTTP request. Figure 3 shows an example page.

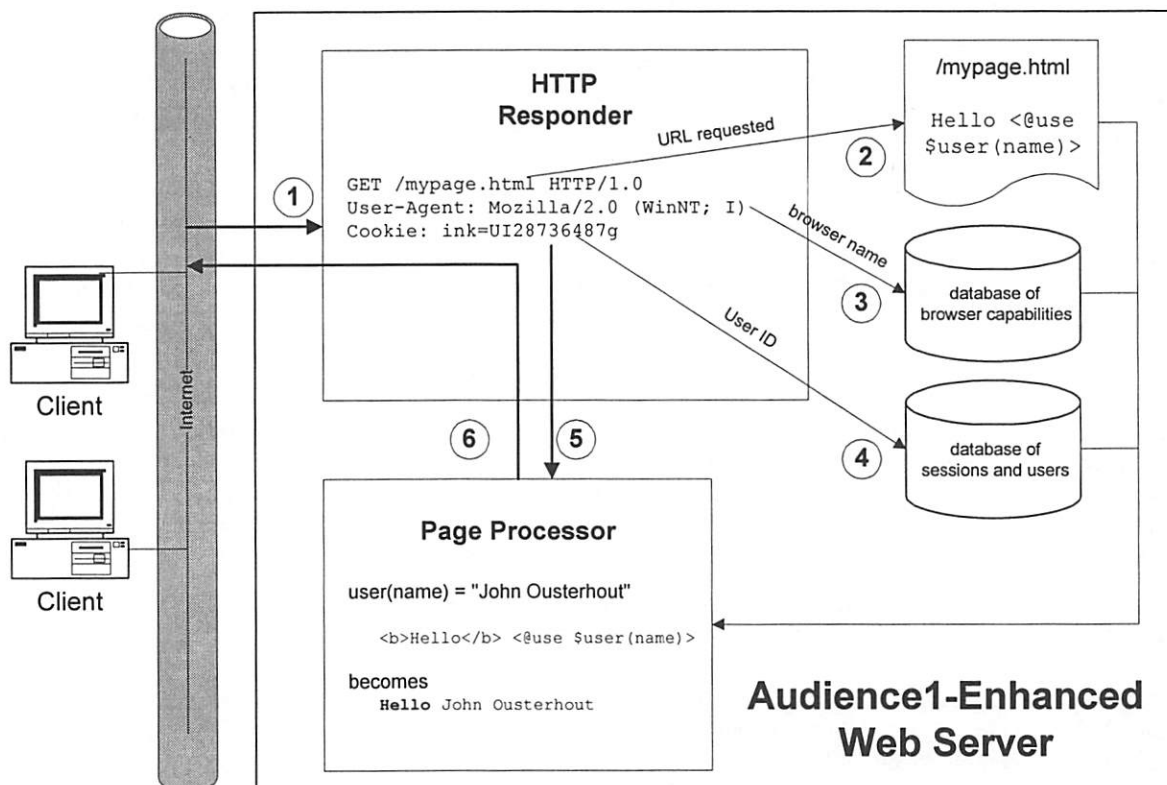


Figure 2. The Audience1 architecture.

(1) At the core of Audience1 is a HTTP responder that listens for incoming requests and assigns a thread to handle each one. (2) Each request is accompanied by a Uniform Resource Locator (URL) request, specifying which (dynamic) HTML page to access, along with a user ID. If the page is valid, the page source text is loaded. (3) The responder provides the User-Agent header (the name of the browser), the client IP address and other HTTP header information to the page processor, processed to be as useful as possible (ie. browser name becomes browser capabilities, IP address becomes location information, such as country). (4) From the user ID, Audience1 links (via a DBMS JOIN) to the user's personal and preference data, if available. Note that user IDs are anonymous unless the end user explicitly attaches it to personal data. (5) This data is then made available to the page processor, whose job it is to sift the HTML for Dynamic Tags™. For each dynamic tag, the processor executes the Tcl code associated with the tag. (6) The output is returned to the user.

V. Tcl's strengths and weaknesses

It is often argued that Tcl's usefulness is tied to the Tk user interface toolkit, yet this example shows a counterexample. Unfortunately, Tcl is not the perfect tool for our purposes. This section explains the pros and cons as we've observed them.

Most of the pros are well-known to the Tcl community: interpretation, everything-is-a-string semantics, easy embedability in C, config files as Tcl scripts and so on. The unusual features we especially appreciated were:

thread safety under Solaris. Our search engine and web server are multi-threaded to allow us to hide the latency of I/O-bound services. This demands a thread-safe scripting language, which MTtcl [Jan95] provides.

lightweight interpreters. MTtcl creates one interpreter per thread; it is imperative that interpreters use a small footprint, which Tcl's do.

low performance risk. Unlike other languages, it is straightforward to extend Tcl from C. This reduces the risk that poor performance will affect product delivery. If a script is too slow, it can be rewritten in C. Note: this technique does not scale (see below).

variable traces. Variable traces on array variables allow us to provide users with the illusion that all accessible data has already been loaded out of the DBMS [Sah95]. In fact, loading data from the DBMS is a heavyweight operation that we would want to cache (memoize). With variable traces, we can load all related data once per HTML page that actually uses that data.

simple syntax. Anyone believing that "syntax doesn't matter" doesn't understand training costs. With Perl or C, it costs a company tens of thousands of dollars per employee in training and lost productivity. The basics of languages such as Scheme and Tcl can be taught to a new user in one day, plus another lost day of productivity fighting with the quoting rules. In developing HotBot, many of our engineers learned Tcl "on the fly" where none found similar success with Perl. Tcl's simple syntax made it feasible to implement Dynamic Tags as Tcl procedure calls. Such an implementation exposes Tcl's syntax to HTML page authors when they substitute variables and call subcommands in the arguments to a dynamic tag.

The following were the weaknesses in Tcl that most affected us:

it's slow. The Berkeley/Sun reference implementation of Tcl is very slow. For HotBot, this becomes a problem when Tcl script processing dominates the time it takes to

return results to the end user, causing us to rewrite many procedures in C. If coding and debugging in C were a fraction as productive as in Tcl, we wouldn't need scripting languages; it is very painful to rewrite more than a few procedures in C. We didn't experiment with the Tcl-to-C compiler from ICEM/CFD; it is unreasonable to ask users to integrate a C compiler into their web publishing process, which would be needed to "compile" dynamic HTML pages statically.

no object support in the core. For both engineering reasons and marketing reasons, object-oriented programming (late binding, type inheritance, implementation inheritance, etc.) is here to stay. Books such as *Design Patterns* [GHJV95] argue that OOP provides an elegant platform for reasoning about code reuse, one of the most successful productivity improvement techniques we know of today [Bro95]. OOP needs to be in the core to be useful: without it, there's little hope that object-based extensions will become widespread. Namespaces (new in Tcl v7.5) are a good first step.

VI. Conclusions

Although the client side of Web programming has been won by Java, there is ample room to pick winners on the server side. We believe that Tcl makes an excellent candidate because of its availability on a plethora of platforms and the number of tools and extensions for it and its embedability.

References

- [Bro95] Frederick P. Brooks. *The Mythical Man Month: 20th anniversary edition* (with new and recent essays). Addison-Wesley. 1995.
- [Broa95] BroadVision Corp. *BroadVision marketing literature*. <http://www.broadvision.com>
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.
- [Jan95] Steve Jankowski. *MTtcl documentation*. <ftp.aud.alcatel.com/tcl/extensions/MTtcl1.0.tar.gz>
- [Nav95] Navisoft Corp. *NaviServer documentation*. <http://www.navisoft.com>
- [Nets96] Netscape Communications Corp. "Persistent Client State: HTTP Cookies". http://home.netscape.com/newsref/std/cookie_spec.html
- [Next95] NeXT Corp. *WebObjects marketing literature*. <http://www.next.com>
- [Sah95] Adam Sah. "Multiple Trace Composition and Its Uses," Proc. Tcl/Tk'95 Symposium. Toronto, Canada. June, 1995.

Writing CGI scripts in Tcl

Don Libes

National Institute of Standards and Technology

libes@nist.gov

Abstract

CGI scripts enable dynamic generation of HTML pages. This paper describes how to write CGI scripts using Tcl. Many people use Tcl for this purpose already but in an ad hoc way and without realizing many of the more non-obvious benefits. This paper reviews these benefits and provides a framework and examples. Canonical solutions to HTML quoting problems are presented. This paper also discusses using Tcl for the generation of different formats from the same document. As an example, FAQ generation in both text and HTML are described.

Keywords: CGI; FAQ; HTML generation; Tcl; World Wide Web

Introduction

CGI scripts enable dynamic generation of HTML pages [BLee]. Specifically, CGI scripts generate HTML in response to requests for Web pages. For example, a static Web page containing the date might look like this:

```
<p>The date is Mon Mar 4 12:50:10 EST
1996.
```

This page was constructed by manually running the date command and pasting its output in the page. The page will show that same date each time it is requested, until the file is manually rewritten with a different date.

Using a CGI script, it is possible to dynamically generate the date. Each time the file is requested, it will show the current date. This script (and all others in this paper) are written in Tcl [Ouster].

```
puts "Content-type: text/html\n"
puts "<p>The date is [exec date]."
```

The first puts command identifies how the browser should treat the remainder of the data – in this case, as text to be interpreted as HTML. For all but esoteric uses, this same first line will be required in every CGI script.

CGI scripts have many advantages over statically written HTML. For example, CGI scripts can automatically adapt to changes in the environment, such as the date in the previous example. CGI scripts can run programs, include and process data, and just about anything that can be done in traditional programs.

CGI scripts are particularly worthwhile in handling Web forms. Web forms allow users to enter data into a page and then send the results to a Web server for processing. The Web form itself does not have to be generated by a CGI script. However, data entered by a user may require a customized response. Therefore, a dynamically generated response via a CGI script is appropriate. Since the response may produce another form, it is common to generate forms dynamically as well as their responses.

CGI Scripts Are Just a Subset of Dynamic HTML Generation

CGI scripts are a special case of generated HTML. Generated HTML means that another program produced the HTML. There can be a payoff in programmatic generation even if it is not demanded by the CGI environment. I will describe this idea further later in the paper.

Simply embedding HTML in Tcl scripts does not in itself provide any payoff. For instance, consider the preparation of a page describing various types of widgets, such as button widgets, dial widgets, etc. Ignoring the body paragraphs, the headers could be generated as follows:

```
puts "<h3>Button Widgets</h3>"
puts "<h3>Dial Widgets</h3>"
```

Much of this is redundant and suggests the use of a procedure such as this one:

```
proc h3 {header} {
    puts "<h3>$header</h3>"
}
```

Now the script can be rewritten:

```
h3 "Button Widget"
```

Reprinted from *The Proceedings of the Fourth Annual Tcl/Tk Workshop '96*, Monterey, CA, July 10-13, 1996.

h3 "Dial Widget"

Notice that you no longer have to worry about adding closing tags such as `/h3` or putting them in the right place. Also, changing the heading level is isolated to one place in each line.

Using a procedure name specifically tied to an HTML tag has drawbacks. For example, consider code that has level 3 headings for both Widgets and Packages. Now suppose you decide to change just the Widgets to level 2. You would have to look at each `h3` instance and manually decide whether it is a Widget or a Package.

In order to change groups of headers that are related, it is helpful to use a logical name rather than one specifically tied to an HTML tag. This can be done by defining an application-specific procedure such as one for widget headers:

```
proc widget_header {heading} {  
    h2 "$header Widget"  
}  
proc package_header {heading} {  
    h3 "$heading Package"  
}
```

The script can then be written:

```
widget_header "Button"  
widget_header "Dial"  
package_header "Object"
```

Now all the widget header formats are defined in one place – the `widget_header` procedure. This includes not only the header level, but any additional formatting. Here, the word “Widget” is automatically appended, but you can imagine other formatting such as adding hyperlinks, rules, and images.

This style of scripting makes up for a deficiency of HTML: HTML lacks the ability to define application-specific tags.

Form Generation

The idea of logical tags is equally useful for generation of Web forms. For example, consider generation of an entry box. Naively rendered in Tcl, a 10-character entry box might look this way:

```
puts "<input name=Username size=10>"
```

This is fine if there is only one place in your code which requires a username. If you have several, it is more convenient to place this in a procedure. Dumping this all into a procedure simplifies things a little, but enough additional attributes on the input tag can quickly render the new procedure impenetrable. Applying the same

technique shown earlier suggests two procedures: `text` and `username`. `text` is the application-independent HTML interface. `username` is the application-specific interface. An example definition for `username` is shown below. Remember that this is specific to a particular application. In this case, a literal prompt is shown (the HTML markup for this would be defined in yet another procedure). Then the 10-character entry box containing some default value.

```
proc username {name defvalue} {  
    prompt "Username"  
    text $name $defvalue 10  
}
```

When the form is filled out, the user's new value will be provided as the value for the variable named by the first parameter, stored here in “name”. Later in this paper, I'll go into this in more detail.

A good definition for `text` is relatively ugly because it must do the hard work of adding quotes around each value at the same time as doing the value substitutions. This is a good demonstration of something you want to write as few times as possible – once, ideally. In contrast, you could have hundreds of application-specific text boxes. Those procedures are trivial to write and make all forms consistent. In the example above, each call to `username` would always look identical.

```
proc text {name defvalue {size 50}} {  
    puts "<input name=\"$name\" \\  
        value=\"$defvalue\" size=$size>"  
}
```

Once all these procedures exist, the actual code to add a username entry to a form is trivial:

```
username new_user $user
```

Many refinements can be made. For example, it is common to use Tcl variables to mirror the form variables. The rewrite in Figure 1 tests whether the named form variable is also a Tcl variable. If so, the value is used as the default for the entry.

If `username` called this procedure, the second argument could be omitted if the variable name was identical to the first argument. For example:

```
username User
```

An explicit value can be supplied in this way:

```
username User=don
```

And arbitrary tags can be added as follows:

```
username User=don size=10 \  
    maxlength=5
```

Many other procedures are required for a full implementation. Here are two more which will be used in the remainder of the paper. The procedure "p" starts a new paragraph and prints out its argument. The procedure "put" prints its argument with no terminating newline. And puts, of course, can be called directly.

```
proc p {s} {
    puts "<p>$s"
}
proc put {s} {
    puts -nonewline "$s"
}
```

Inline Directives

Some HTML tags affect characters rather than complete elements. For example, a word can be made bold by surrounding it with and . As before, redundancy can be eliminated by using a procedure:

```
proc bold {s} {
    puts "<b>$s</b>"
}
```

Unlike the earlier examples, it is not desirable to have character-based procedures call puts directly. Otherwise, scripts end up looking like this:

```
put "I often use "
bold "Tcl"
put "to program."
```

These character-based procedures can be made more readable by having them return their results like this:

```
proc bold {s} {
    return "<b>$s</b>"
}
```

Using these inline directives, scripts become much more readable:

```
p "I often use [bold Tcl] to
program."
```

Explicit use of a procedure such as bold shares the same drawbacks as explicit use of procedures such as h2 and h3. If you later decide to change a subset of some uses,

```
proc text {nameval args} {
    regexp "([^\=]*) (=?) (.*)" $nameval dummy name q value
    put "<input name=\"${name}\""
    if {$q != "="} {
        set value [uplevel set $name]
    }
    puts " value=\"[quote_html $value]\" $args"
}
```

Figure 1: Procedure to create a generic text entry

you must examine all of them. By using logical procedure names, that trap is avoided. For example, suppose that you want hostnames to always appear the same way. But there is no hostname directive in HTML. So you could arbitrarily choose bold and write:

```
proc hostname {s} {
    return [bold $s]
}
```

An example using this is:

```
p "You may ftp the files from [host
$ftphost] or [host $ftpbakuphost]."
```

If you later decide to change the appearance of hostnames to, say, italics, it is now very easy to do so. Simply change the one-line definition of the hostname procedure.

URLs

URLs have a great deal of redundancy in them, so using procedures can provide dramatic benefits in readability and maintainability. Similarly to the previous section, hyperlinks can be treated as inline directives. By pre-storing all URLs, generation of a URL then just requires a reference to the appropriate one. While separate variables can be used for each URL, a single array (_cgi_link) provides all URL tags with their own namespace. This namespace is managed with a procedure called link. For example, suppose that you want to produce the following display in the browser:

```
I am married to Don Libes who works in the
Manufacturing Collaboration Technologies
Group at NIST.
```

Using the link procedure, with appropriate link definitions, the scripting to produce this is simple:

```
p "I am married to [link Libes] who
works in the [link MCTG] at [link
NIST]."
```

This expands to a sizeable chunk of HTML:

```
I am married to <A HREF="http://
elib.cme.nist.gov/msid/staff/libes/
libes.don.html">Don Libes</A> who
works in the <A HREF="http://
elib.cme.nist.gov/msid/groups/
mctg.htm">Manufacturing
Collaboration Technologies Group</A>
at <A HREF="http://
www.nist.gov">NIST</A>.
```

Needless to say, working on such raw text is the bane of HTML page maintainers. Yet HTML has no provisions itself for reducing this complexity.¹

The link procedure is shown in Figure 2. It returns the formatted link given the tag name as its first argument.

1. It is tempting to think that relative URLs can simplify this, but relative URLs only apply to URLs that are, well, relative. In this example, the URLs point to a different host than the one where the referring page lives. Even if this isn't the case, I avoid relative URLs because they prevent other people from copying the raw HTML and pasting it into their own page (again, on another site) without substantial effort in first making the URLs absolute.

```
proc link {args} {
    global _cgi_link

    set tag [lindex $args 0]
    if {[llength $args] == 3} {
        set _cgi_link($tag) \
            "<A HREF=\"[lindex $args 2]\">[lindex $args 1]</A>"
    }
    return $_cgi_link($tag)
}
```

Figure 2: Procedure to access a database of URL links.

```
set MSID_STAFF $MSID_HOST/msid/staff

link Steve      "Steve Ray"      $MSID_STAFF/ray.steve.html
link Don        "Don Libes"      $MSID_STAFF/libes.don.html
link Josh       "Josh Lubell"    $MSID_STAFF/josh.lubell.html
```

Figure 3: Create links to several colleagues whose home pages all exist in the same staff directory.

```
set MSID_HOST http://elib.cme.nist.gov
set NIST_HOST http://www.nist.gov
set ORA_HOST  http://www.ora.com
```

Figure 4: Some examples of hosts.

```
link Don        "Don"            $MSID_STAFF/libes.don.html
link Libes      "Don Libes"      $MSID_STAFF/libes.don.html
```

Figure 5: Create links to the same URL but display them to the user differently.

The second argument, if given, declares a name to be displayed by the browser. The third argument is the URL.

Links can be defined by handcoding the complete absolute URL. However, it is much simpler to create a few helper variables to further minimize redundancy. Figure 3 shows how to refer to several of my colleagues whose home pages all exist in the same staff directory.

If the location of any one staff member's page changes, only one line needs to be changed. More importantly, if the directory for the MSID staff pages changes, only one line needs to be changed. MSID_STAFF is dependent on another variable that defines the hostname. The hostname is stored in a separate variable because 1) it is likely to change and 2) there are other links that depend on it.

Figure 4 shows some examples of hosts.

There are no restrictions on tag names or display names. For example, sometimes it is useful to display "Don". Sometimes, the more formal "Don Libes" is appropriate. This is done by defining two links with different names but pointing to the same URL. This is shown in Figure 5.

Similarly, there are no restrictions on the tag names themselves. Consider the link definitions in Figure 6. These are used in paragraphs such as this one:

```
p "You can ftp Expect from
ftp.cme.nist.gov as [link Expect.Z]
or [link Expect.gz]"
```

A browser shows this as:

You can ftp Expect from ftp.cme.nist.gov as
pub/expect/expect.tar.Z or ...gz.

Having link dependencies localized to one place greatly aids maintenance and testing. For example, if you have a set of pages that use the definitions (i.e., by sourcing them), editing that one file automatically updates all of the other pages the next time they are regenerated. This is useful for testing groups of pages on a different server, such as a test server before moving them over to a production location. Even smaller moves can benefit. For example, it is common to move directories around or create new directories and just move some of the files around.

Quoting

HTML values must be quoted at different times and in different ways. Unfortunately, the standards are hard to read so most people guess instead. However, intuitively figuring out the quoting rules is tricky because simple cases don't require quoting and many browsers handle various error cases differently. It can be very difficult to deduce what is correct when your own browser accepts erroneous code. This section presents procedures for handling quoting.

CGI Arguments

CGI scripts can receive input from either forms or URLs. For example, in a URL specification such as `http://www.nist.gov/expect?help=input+foo`, anything to the right of the question mark becomes input to the CGI script (which conversely is to the left of the question mark).

Various peculiar translations must be performed on the raw input to restore it to the original values supplied by the user. For example, the user-supplied string "foo bar" is changed to "foo+bar". This is undone by the first `regsub` in `unquote_input` (shown in Figure 7). The remain-

ing conversions are rather interesting but understanding them is outside the point of this paper.

The converse procedure to `unquote_input` is shown below. This transformation is usually done automatically by Web browsers. However, it can be useful if your CGI script needs to send a URL through some other means such as an advertisement on TV.

```
proc quote_url {in} {
    regsub -all " " $in "+" in
    regsub -all "%" $in "%25" in
    return $in
}
```

In theory, this procedure should perform additional character translations. However, you should avoid generating such characters since receiving URLs outside of a browser requires hand-treatment by users. In these situations, all bizarre character sequences should be avoided. For the purposes of testing (feeding input back), additional translation is also unnecessary since any other unquoted characters will be passed untouched.

Suppressing HTML Interpretation

In most contexts, strings which contain strings that *look* like HTML will be interpreted as HTML. For example, if you want to display the literal string "", it must be encoded so that the "<" is not turned into a hyperlink specification. Other special characters must be similarly protected. This can be done using `quote_html`, shown below:

```
proc quote_html {s} {
    # ampersand must be done first!
    regsub -all {&} $s {\&} s
    regsub -all {"} $s {\"} s
    regsub -all {<} $s {\<} s
    regsub -all {>} $s {\>} s
    return $s
}
```

This can be used to simplify other procedures. Adding explicit double quotes before returning the final value allows simplification of many other procedures. Assuming this new procedure is called `dquote_html`, consider the earlier text entry procedure which had the code fragment

```
value="\$defvalue"
```

This could be rewritten:

```
value=[dquote_html $defvalue]
```

```
link Expect.Z      "pub/expect/expect.tar.Z"  $EXPECT_DIR/expect.tar.Z
link Expect.gz     "...gz"                    $EXPECT_DIR/expect.tar.gz
```

Figure 6: Link tags and definitions can be very unusual. There are no restrictions.

Argument Cracking

As described earlier, input strings to a CGI script are encoded by the browser. Besides the transformations described already, the browser also packs all variable values together in the form `variable1=value1&variable2=value2&variableN=valueN`.

The input procedure (Figure 8) splits the input back into its specific variable/value pairs leaving them in a global array called `_cgi_var`. Any variable ending with the string "List" causes its value to be treated as a Tcl list. This allows, for example, multiple elements of a listbox to be extractable as individual elements.

If the procedure is run in the CGI environment (i.e., via an HTTPD server), input is automatically read from the environment. If not run from the CGI environment (i.e., via the command line), the argument is used as input. This is very useful for testing. An explicit argument obviates the need for using a real form page to drive the script and means it is easily run from the command line or a debugger.

If the global variable `_cgi(debug)` is set to 1, the procedure prints the input string before doing anything else. This is useful because it may then be cut and pasted into the procedure argument for debugging purposes, as was just mentioned.

```
proc unquote_input {buf} {
    # rewrite "+" back to space
    regsub -all {\+} $buf {\ } buf

    # protect $ so Tcl won't do variable expansion
    regsub -all {\$} $buf {\$} buf

    # protect [ so Tcl doesn't do evaluation
    regsub -all {\[} $buf {\[} buf

    # protect quotes so Tcl doesn't terminate string early
    regsub -all \" $buf \\\\" buf

    # replace line delimiters with newlines
    regsub -all %0D%0A $buf %0A buf
    # Mosaic sends just %0A. This is handled in the next command.

    # prepare to process all %-escapes
    regsub -all {%([A-F0-9][A-F0-9])} $buf {[format %c 0x\1]} buf
    # Mosaic sends just %0A. This is handled in the next command.

    # prepare to process all %-escapes
    regsub -all {%([A-F0-9][A-F0-9])} $buf {[format %c 0x\1]} buf

    # process %-escapes and undo all protection
    eval return \"$buf\"
}
```

Figure 7: Translate HTML-style input to original data.

Import/Export

Variables are not automatically entered into separate global variables or the `env` array because that would open a security hole. Instead, variables must be explicitly requested. Several procedures simplify this. The procedure most commonly used is "import".

`import` is called for each variable defined from the invoking form. For example, if a form used an entry with "name=foo", the command "import foo" would define `foo` as a Tcl variable with the value contained in the entry. The command `import_cookie` is a variation that obtains the value from a cookie variable – a mechanism that allows client-side caching of variables.

```
proc import {name} {
    upvar $name var
    upvar #0 _cgi_uservar($name) val

    set var $val
}
```

Form variables are automatically exported to the called CGI script. It is sometimes necessary to export other variables. This must be done explicitly. Figure 9 shows the export procedure which exports the named variable. Similar to the text procedure, if the first argument is in the form "var=value", the variable is exported with the given value. Otherwise, the variable is treated as a Tcl variable and its value is used.

Error Handling

The CGI environment makes no special provisions for errors. Thus, error processing requires explicit handling by the application programmer. If none is made, any error messages produced (e.g., by the Tcl interpreter) are sent on to the client browser. These are rarely meaningful to the user. Even worse, they can be misinterpreted as HTML in which case the result is incomprehensible even to the script creator.

The procedure in Figure 10 provides a framework to evaluate the body of the CGI script, to automatically

catch errors, and attempt to do something useful. The two arguments, head and body, are blocks of Tcl commands which create the head and body of an HTML form. An example is shown later.

If the global value `_cgi(debug)` is 1, the script error is formatted and printed to the screen so that it is readable. If debug is 0, a simple message is printed saying that an error occurred and that the "diagnostics are being emailed to the service system administrator". At the same time, mail is sent to the service administrator. The mail includes everything about the environment that is necessary to reproduce the problem including the error,

```
proc input {{fakeinput {}}} {
    global env _cgi _cgi_uservar

    if {[info exists env(REQUEST_METHOD)]} {
        set input $fakeinput;# running by hand, so fake it
    } elseif { $env(REQUEST_METHOD) == "GET" } {
        set input $env(QUERY_STRING)
    } else {
        set input [read stdin $env(CONTENT_LENGTH)]
    }
    # if script blows up later, enable access to the original input.
    set _cgi(input) $input

    # good for debugging!
    if {$_cgi(debug)} {
        puts "<pre>$input</pre>"
    }

    set pairs [split $input &]
    foreach pair $pairs {
        regexp (.*)=(.*) $pair dummy varname val

        set val [unquote_input $val]

        # handle lists of values correctly
        if [regexp List$ $varname] {
            lappend _cgi_uservar($varname) $val
        } else {
            set _cgi_uservar($varname) $val
        }
    }

    # repeat loop above but for cookies
}
```

Figure 8: Retrieve CGI input

```
proc export {nameval} {
    regexp "([^\=]*)=(?)(.*)" $nameval dummy name q value

    if {$q != "="} {
        set value [uplevel set $name]
    }

    put "<input type=hidden name=$name \
        value=[dquote_html $value]>"
}
```

Figure 9: Export a variable to the CGI script.

the script name, and the input. The implementation shown here is skeletal. In the actual definition, a variety of other interesting problems are handled. For instance, cookie definitions must appear in the output before any HTML. However, cookies are more easily generated as one of the final results in a script. This and other problems are solved by the full implementation, however the details are beyond the scope of this paper.

Using the procedures defined, CGI scripts become very simple. They all start out by sourcing the CGI support routines. Then `cgi_eval` is called with arguments to create the head and body. The head generates titles, link

colors, etc., while the body is responsible for importing, exporting, and generation of text and graphical elements as has already been described. A skeletal example is shown in Figure 11

The title procedure (not shown) produces all of the usual HTML boilerplate including titles, backgrounds, etc. A form procedure simplifies the calling conventions for establishing any forms. This is not difficult. However, of critical importance is noting that a form is in progress. Because some browsers won't show anything if a form hasn't been ended (i.e., `</form>`), the error handler must prematurely close the form if an unexpected

```
proc cgi_eval {head body} {
    global env _cgi

    set _cgi(body) "$head;cgi_body_start;app_body_start;$body;app_body_end"
    uplevel #0 {
        cgi_body_start
        if 1==[catch $_cgi(body)] {          # errors occurred, handle them
            set _cgi(errorInfo) $errorInfo

            # close possible open form because some
            # browsers won't show errors otherwise
            if [info exists _cgi(form_in_progress)] {
                puts "</form>"
            }

            h3 "An internal error was detected in the service software. \
                The diagnostics are being emailed to the service\
                system administrator."

            if {$_cgi(debug)} {
                puts "Heck, since you're debugging, I'll show you the\
                    errors right here:"
                # suppress formatting
                puts "<xmp>$_cgi(errorInfo)</xmp>"
            } else {
                mail_start $_cgi(email_admin)
                mail_add "Subject: $_cgi(name) problem"
                mail_add
                if {$env(REQUEST_METHOD) != "by hand"} {
                    mail_add "CGI environment:"
                    mail_add "REQUEST_METHOD: $env(REQUEST_METHOD)"
                    mail_add "SCRIPT_NAME: $env(SCRIPT_NAME)"
                    catch {mail_add "HTTP_USER_AGENT: $env(HTTP_USER_AGENT)"}
                    catch {mail_add "REMOTE_ADDR: $env(REMOTE_ADDR)"}
                    catch {mail_add "REMOTE_HOST: $env(REMOTE_HOST)"}
                }
                mail_add "input:"
                mail_add "$_cgi(input)"
                mail_add "errorInfo:"
                mail_add "$_cgi(errorInfo)"
                mail_end
            }
        }
        cgi_body_end
    }
}
```

Figure 10: Framework to catch errors and report them intelligently.

error occurs. Saving this information is done with a simple global variable. The form procedure is shown in Figure 12.

Many other utilities are necessary such as procedures for each type of form element. Space prevents inclusion of them. Several other miscellaneous utilities complete the basic implementation of the procedures that appear in this paper. A few are mentioned here to give a flavor for what is necessary:

cgi	Converts a form name to a complete URL.
mail_start	Generates headers and writes them to a new file representing a mail message to be sent.
mail_add	Writes a new line to the temporary mail file.
mail_end	Appends a signature to the temporary mail file, sends it, and deletes the file.
cgi_body_start	Generates the <body> tag and handles user requests such as backgrounds and various color

options. cgi_body_end is analogous.

All of the procedures described so far can be invoked with "cgi_" prepended (if they do not already begin that way). In practice, CGI scripts are generally quite short so this isn't often useful – and writing things like "cgi_h2" is particularly irritating. However conflicts with other namespaces can occasionally make such prefixes a necessary evil.

Several procedures are expected to be redefined by the user. Here are two examples that appear in the body procedure earlier.

app_body_start	Application-supplied procedure, typically for writing initial images or headers common to all pages.
app_body_end	Application-supplied procedure, typically for writing signature lines, last-update-by, etc.

```
source cgi.tcl

cgi_eval {
  title "Password Change Acknowledgment"
  input "name=libes&old=swordfish&new1=tgif23&new2=tgif23"
} {
  import name
  import old

  ... other stuff
  form password {

    spawn /bin/passwd
    expect "Password:"
    ...
  }
}
```

Figure 11: Skeletal example of the CGI procedures in use.

```
proc form {name cmd} {
  global _cgi

  set _cgi(form_in_progress) 1
  puts "<form method=POST action=[cgi $x]>"
  uplevel $cmd
  puts "</form>"
  unset _cgi(form_in_progress)
}
```

Figure 12: The form procedure creates an HTML-style form.

FAQ generation

Earlier I mentioned that CGI scripts are just a subset of HTML generation. As an example, consider the task of building an FAQ in HTML. There is no benefit to dynamically generating an FAQ – it rarely changes. However, an FAQ has some of the same problems as I described earlier. For example, it can include many links which must be kept current.

Another reason that it makes sense to think about generating HTML for an FAQ is that an FAQ is highly stylized. For example, an FAQ always has a set of questions. These questions are then repeated but with answers. Written manually, you would have to literally repeat the questions and create the links. If a new question was added or an old one deleted, you would have to carefully make sure that both entries were handled identically.

Intuitively, this could be automated using two loops. First, the questions and answers would be defined. Then the first loop would print the questions. The second loop would print the questions (again) interspersed with the answers. In pseudocode:

```
define QAs                ;# pseudocode!

foreach qa $QAs {
    print_question $qa
}

foreach qa $QAs {
    print_question $qa
    print_answer $qa
}
```

It suffices to store the questions and answers in an array. The following code numbers each pair and stores ques-

tion N in qa(N,q) and the corresponding answer in qa(N,a). At the same time, the question is printed out. Thus, there is no need for the first loop in the earlier pseudocode.

```
proc question {q a} {
    global index qa

    incr index

    set qa($index,q) $q
    set qa($index,a) $a

    puts "<A HREF=\"#$q$index\">"
    puts "<li>$q"
    puts "</A>"
}
```

Each question automatically links to its corresponding answer, linked as #qN. When the question/answer pairs are later printed, they will have A HREF tags defining the #qN targets.

The source for an example question/answer definition is shown in Figure 13.

The question is now only stated once and it is always paired with the answer. This simplifies maintenance.

Notice that the answer is not simply a string. The answer is Tcl code. This makes it possible to use all of the techniques mentioned earlier. For example, the example above uses p to generate new paragraphs and link to generate hyperlinks.

The code is evaluated by passing the answer to eval whenever it is needed. An answer procedure does this and generates the hyperlink target at the same time.

```
proc answer {i} {

    question {I keep hearing about Expect. So what is it?} {

        p "Expect is a tool primarily for automating interactive applications
        such as telnet, ftp, passwd, fsck, rlogin, tip, etc. Expect really
        makes this stuff trivial. Expect is also useful for testing these
        same applications. Expect is described in many books, articles,
        papers, and FAQs. There is an entire [link book] on it available
        from [link ORA]."

        p "You can ftp Expect from ftp.cme.nist.gov as [link Expect.Z] or
        [link Expect.gz]"

        p "Expect requires Tcl. If you don't already have Tcl, you can get
        it in the same directory (above) as [link Tcl.Z] or [link Tcl.gz]."

        p "Expect is free and in the public domain."

    };# end question
}
```

Figure 13: Source to an example question/answer definition.


```

global qa

puts "<p>"
puts "<A NAME=\"q$i\">"
puts "<li><b>$qa($i,q)</b>"
puts "</a>"
puts "<p>"
eval $qa($i,a)
}

```

For example, "answer 0" would produce the beginning of the output from the earlier question. The full HTML would begin like this:

```

<p>
<A NAME="q0">
<li><b>I keep hearing about Expect.
So what is it?</b>
</a>
<p>Expect is a tool primarily for
automating interactive . . .

```

The answer procedure itself is called from a loop in another procedure called answers (Figure 14). An answer_header procedure prints out a header if one has been associated with the current question. This provides a way of breaking the FAQ into sections. A matching procedure (question_header) defines and prints the headers as they are encountered.

```

proc answer_header {i} {
    global qa

    h3 "$qa($i,h)"
}

proc question_header {h} {
    global index qa

    set qa($index,h) $h
    puts "<A HREF=\"#h$index\">"
    h3 $h
    puts "</A>"
}

```

Translation to Other Formats

Another benefit of using logical tags is that different output formats can be generated by changing the appli-

cation-specific procedures. For instance, suppose a horizontal rule is produced using the hr command. Obviously this can be defined as "puts <hr>". It is easily changed to produce text using the following procedure:

```

proc hr {} {
    puts =====
}

```

Here are analogous definitions for h1 and h2. Others are similar.

```

proc h1 {s} {
    puts ""
    puts ""
    puts "" $s
    puts ""
    puts ""
}

proc h2 {s} {
    puts "" $s ""
}

```

For example, with this new definition, "h1 Questions" reasonably simulates a level 1 header using only text as:

```

*
* Questions
*

```

The ability to generate the FAQ in different forms is convenient. For example, it means that people can read the FAQ without having an HTML browser.

The generation of different formats is simplified by avoiding use of explicit HTML tags and instead using logical procedure names. A particular output format can be produced merely by providing an appropriate set of procedure definitions. Although I have not done so, it should be possible to adapt the framework and ideas shown here to produce output in such formats as TEX, MIF, and others. Even without translation, avoiding explicit HTML is a good idea for the reasons mentioned earlier – maintenance and readability.

```

proc answers {} {
    uplevel #0 {
        start_answers
        for {set index 0} {$index < $maxindex} {incr index} {
            catch {answer_header $index}
            answer $index
            hr
        }
    }
}

```

Figure 14: Generate all the answers in the FAQ.

A Translation Framework

Translation is further simplified by separating the application-specific definitions from the content of the particular document. For example, multiple FAQs could reuse the same set of FAQ support definitions. Each FAQ would start by loading the FAQ definitions by means of a source command appropriate to the desired output:

```
source FAQdriver.$argv
```

A driver for each output format defines the procedures to produce the FAQ in that particular format. For example, FAQdriver.html would begin:

```
# driver.html - Tcl to HTML procs
proc hr {} {puts "<hr>"}
```

FAQdriver.text would start similarly:

```
# driver.text - Tcl to text procs
proc hr {} {puts "=====}"
```

If short enough, all of the different definitions can be maintained as a single file which simply uses a switch to define the appropriate definitions.

```
switch $argv {
  html {
    proc emphasis {s} {
      puts "<em>$s</em>"
    }
    . . .
  }
  text {
    proc emphasis {s} {puts "$s"}
    . . .
  }
}
```

In either case, output generation is then accomplished by executing the document with the argument describing the desired format. For example, assuming the FAQ source is stored in ExpectFAQ, HTML is generated from the command line as:

```
% ExpectFAQ html
```

Text output is generated as:

```
% ExpectFAQ text
```

Experiences

The techniques described in this paper have been used successfully in building several projects consisting of large numbers of pages including the NIST Application Protocol Information Base [Lubell] and the NIST Identifier Collaboration Service [Libes95]. In addition, they have been used to construct and maintain several FAQs including the Expect FAQ [Libes96].

Readers interested in comparative strategies to CGI generation should consult the Yahoo database [Yahoo] which lists CGI libraries for dozens of languages, often with multiple entries for each. Readers should also explore alternative strategies to CGI, such as the Tcl-based server-side programming demonstrated by Audience1 [Sah] and NeoScript [Lehen] which elegantly solve problems that CGI alone cannot address adequately.

The other aspect of this paper, dynamic document generation, is also an area rich in development. Various attempts are being made to solve this in other ways including SGML and its extensions and alternatives. Good discussion of these can be found in [Harman].

Concluding Notes

This paper has shown the benefits of generating HTML from Tcl scripts. CGI scripts are an obvious use of this. However, even static documents benefit by increasing readability and improving maintainability.

Traditionally, Perl has been the language of choice for CGI scripting. However, use of Tcl for CGI scripting has increased significantly. Part of this is simply due to the number of people who already know Tcl. But Tcl brings with it many beneficial attributes: Tcl is a simple language to learn. Its portability is excellent, it is robust, and it has no significant startup overhead. And of course it is easily embeddable in other applications making it that much easier to leverage ongoing development in languages such as C and C++.

These are all characteristics that make Tcl very attractive for CGI scripting. However, Tcl does not have a history of use for CGI scripting and there is little documentation to help beginners get started. Hopefully, this paper will make it easier for more people to get starting writing CGI scripts in Tcl.

Availability

The CGI library described is available at <http://www.cme.nist.gov/pub/expect/cgi.tcl.tar.Z>. The FAQ library described can be retrieved from the Expect FAQ itself [Libes96]. This software is in the public domain. NIST and I would appreciate credit if you use this software.

Acknowledgments

Thanks to Josh Lubell, John Buckman, Mark Williamson, Steve Ray, and the Tcl '96 program committee for valuable suggestions on this paper.

References

- [BLee] T. Berners-Lee, D. Connolly, "Hypertext Markup Language – 2.0, RFC 1866, HTML Working Group, IETF, Corporation for National Research Initiatives, URL: http://www.w3.org/pub/WWW/MarkUp/html-spec/html-spec_toc.html, September 22, 1995.
- [Harman] Harman, D., "Overview of the Third Text REtrieval Conference (TREC-3), NIST Special Publication 500-225, NIST, Gaithersburg, MD, April 1995.
- [Lehen] Lehenbauer, K., "NeoScript", URL: <http://www.NeoSoft.com/neoscript/>, 1996.
- [Libes95] Libes, D., "NIST Identification Collaboration Service", URL: <http://www-i.cme.nist.gov/cgi-bin/ns/src/welcome.cgi>, National Institute of Standards and Technology, 1995.
- [Libes96] Libes, D., "Expect FAQ", URL: <http://www.cme.nist.gov/pub/expect/FAQ.html>, National Institute of Standards and Technology, 1996.
- [Lubell] Lubell, J., "NIST Identification Collaboration Service", URL: <http://www-i.cme.nist.gov/proj/apde/www/apib.htm>, National Institute of Standards and Technology, 1996.
- [Ouster] Ousterhout, J., "Tcl and the Tk Toolkit", Addison-Wesley Publishing Co., 1994.
- [Sah] Sah, A., Brown, K., and Brewer, E., "Programming the Internet from the Server-Side with Tcl and Audience1", Tcl/Tk Workshop 96, Monterey, CA, July 10-13, 1996.
- [Yahoo] "Yahoo!", URL: http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI___Common_Gateway_Interface/, April, 1996.

Lessons from the Neighborhood Viewer: Building Innovative Collaborative Applications in Tcl and Tk

Alex Safonov, Douglas Perrin, Joseph Konstan, John Carlis, John Riedl
Department of Computer Science

Robert Elde
*Department of Cell Biology and Neuroanatomy
University of Minnesota*

{safonov,dperrin,konstan,carlis,riedl}@cs.umn.edu, elde@med.umn.edu

Abstract

This paper discusses the development in Tk of a collaborative browser for scientific image databases. The browser, known as the "neighborhood viewer," allows groups of neuroscientists to explore systematically a large collection of brain images. The paper discusses the application, its development, and a set of lessons learned during development. In particular, it shows how constraints and distributed constraints simplified development, discusses the implementation of a wavelet-based image format, and draws lessons about engineering experimental, evolving systems.

This paper tells the story of a group of scientists working together to build an environment in which many people could work together to study the brain. There are many challenges for neuroscientists in brain research, but we focus on the lessons that computer scientists learned as we used Tcl and Tk to invent new interfaces and applications. Our efforts bear fruit not only in the neighborhood viewer application (pictured in Figure 1) but also in a set of experiences and lessons that will make it easier for us to develop other innovative applications. We hope the reader will benefit from our experiences and thereby develop applications faster and more creatively as well.

We structure this paper as a development narrative. The next section gives the background and outlines the story. The following three sections focus more closely on three of the major lessons: lessons on the use of constraint programming, flexible coupling through distributed constraints, and the incorporation of wavelet-compressed images. The final section presents general lessons about engineering innovative and complex applications using Tcl and Tk.

The Neighborhood Viewer

Once upon a time... The context of this story is a larger, interdisciplinary project to extend database management system (DBMS) technology to support, in an integrated fashion, large volumes of neuroscience data from structural, functional, and behavioral studies using rodents. We intend to integrate neuroscience at the data level and thereby both enable cross-disciplinary neuroscience research and drive applied computer science research [1, 3]. While our initial focus was dealing with structural data (primarily images obtained from several different modalities, e.g., confocal, laser microscope and digital camera), it quickly became apparent that we had an opportunity to build novel user interfaces. This story describes their evolution.

As we proceeded, it became clear that no one of us was learned enough in both neuroscience and computing to develop these applications. Accordingly, our paradigm required two crucial elements:

mutual education We needed to educate each other. While computer scientists were already building scientific databases [7, 2], and neuroscientists were computer literate, there was much vocabulary to learn just so we could talk to each other. Also computer scientists needed to learn what neuroscience was about - issues addressed and ways of neuroscientific work. Notable was that neuroscience work changes as new imaging technology becomes available. Neuroscientists needed to learn ways of computer science work and to get a sense of the possible.

joint invention We worked together to invent interfaces. Several times we iterated through a two stage process. First, computer scientists observed neuroscientists doing neuroscience,

and then jointly held brainstorming sessions. Second, the computer scientists designed and built a prototype, brainstormed on the prototype's interface details among themselves, and then jointly examined it with the neuroscientists. All were stimulated to invent new interface features.

These elements occurred simultaneously. Indeed, we did not begin to truly understand each other until our minds met at a prototype's interface. Note that we did not prototype as a means of users transmitting known, but unarticulated requirements. What we did was invent new tools, and, thus, new ways to do neuroscience. Note also that a) we did not try to have each prototype support every valuable feature, b) while eventually much of our data will be stored by a DBMS, we did not insist on using the DBMS for prototyping, and c) the later iterations described next really did overlap somewhat, but the story is simpler to tell if we ignore that.

In the first iteration computer scientists found out that confocal images a) are captured in modest-sized rectangles (about 500x750 pixels) applying 1 to 3 wavelength filters; b) are, in a labor intensive fashion, beginning to be "montaged" into larger rectangles. Confocal microscopy allows to capture many images at varying depths (z values) from one specimen. We characterized the data as "locally dense, but globally sparse" - most neuroscientific work focuses on a few "named brain locations." Neuroscientists are interested in the spatial juxtaposition of, for example, opioid receptors and acceptors, each appearing under a different filter. A simple viewer was built that had straightforward pan and zoom features, a bookmark feature, and, mimicking what happens at the microscope, could "flicker" between the part of an image captured with different filters. Together we decided that:

- Monitor real estate is precious and that the interface should allow the user to toggle off temporarily unneeded buttons, sliders, etc.
- One needs to be able to go forward or back through a stack of images, which we termed "rolodexing."
- While pan and zoom are needed and good for answering the question "what is this?", they are insufficient for the question "where am I?" A zoom out followed by a directed zoom in does work for knowing where one is within an x,y image, but is not sufficient for knowing

where one is within the brain in the (relative) z dimension.

- It is a labor intensive and expensive task to print out an image, mark interesting spots and record comments. Doing these tasks on-line became a requirement.

It was not hard for us to decide at that point to implement in Tcl/Tk, because a) we were already comfortable with Tcl/Tk, and b) we did not know where the interface would evolve.

In the second iteration computer scientists learned that images were generally from a coronal cutting plane, but that sagittal and horizontal cutting planes (respectively: vertical - front to back; vertical - side to side; and horizontal - top to bottom) are also valuable. They observed neuroscientists, to help themselves determine "where am I," using a rat brain atlas consisting of coronal images with a visual index: a sagittal silhouette plus a "here" vertical line. Computer scientists saw expert neuroscience researchers using the atlas much less than graduate students. Together we decided that:

- Users should be able to simultaneously display related images from stacks in different cutting planes - generalizing the atlas's silhouette. The intersection point of the image planes defines a "neighborhood."
- Movement (pan or rolodex) in one image causes a redefinition of the neighborhood and should induce appropriate "yoked" movement in other images.
- To aid in answering "where am I?" users should have the option to display surrounding, "nearby" images of the current focal image in a stack.
- For yoked movement to occur users must a priori assert how far apart in (relative) z images within a stack are, and how pairs of stacks line up.
- Users must be able to temporarily unyoke and then reyoke.

As we added more and varied interface elements, the computer scientists found themselves bogged down. Even with Tcl/Tk our code was becoming too intricate, and implementing got progressively more difficult. This motivated us to consider the constraint model.

- Display, at first, only fuzzy images, which are sharpened on demand. (This can be done relatively quickly)
- Allow a user to see nearby images at the same or fuzzier precision.

Image processing costs led us to search for an alternate solution to these two requirements. Wavelet analysis offered a solution which met both. We found that as little as .2% of the data is sufficient for coarse “where am I” decisions, while 2% serves for finer ones. For most “what is this” questions 5-10% often works and only occasionally is full detail needed.

In the next sections we present lessons learned in using constraints, distributed constraints and wavelets, overall lessons, and conclusions.

Programming with Constraints

Imagine a neuroscientist browsing brain data. Multiple 2D views of data are displayed; each is a projection centered on the same point in the brain. As the neuroscientist navigates in any of the views, the other views should update their images to remain consistent, i.e., centered on the same point. No matter how navigation is done - by dragging the image, scrolling, or directly entering new center coordinates, all views must remain consistent. Imagine also that the neuroscientist wants to display in the horizontal view two additional images that are 10 microns above and below the focal image. In our implementation of such a multi-view browser we would like to avoid two problems: having each view explicitly update all other ones, and remember which views are added and deleted.

The constraint-based programming model offers a good solution. This model allows the programmer to specify relationships between variables explicitly, independent of the control flow. The constraint engine enforces these relationships at runtime by updating dependent variables whenever related source variables change. These relationships are called formulas or constraints.

Our original implementation did not use constraints. Scrollbars were used to pan in x and y directions, and a scale allowed users to select another image from the stack. When moved, a scrollbar and scale widget sent the changed center coordinates to all scrollbars and scales in other views. We found our code hard to read and modify. We therefore chose to use TclProp [5] to re-implement the

neighborhood viewer. With constraints, we identified a global 3D position and encapsulated each view's state in the x, y and z coordinates of its center. Since each view is the projection of the 3D space, we found it intuitive to express the x, y and z coordinates of its center as transformations of the global 3D position. The differences in data communication and Tcl code were significant and are illustrated in Figure 2.

The encapsulation of the view coordinates gave us the two important benefits. When the neuroscientist navigates in a view, for example, by moving a scrollbar, the scrollbar just sets the coordinates of the center appropriately. The formulas ensure that global position, and variables storing center coordinates of other views, are updated. When a view's coordinate variables change as a result of formula propagation, the constraints linking them to control widgets are triggered, and update the image in the view. For example, constraints on x and y of the view center scroll the view window in the x or y direction, respectively. The constraint on z loads a new image corresponding to the new value of z. The code became a lot simpler, since coordinate transformations were encapsulated in formulas between local and global coordinate variables, and did not have to be specified for each scrollbar or scale widget.

Second, with constraints, widget callbacks did not have to know about widgets in other views. Scrollbars and scales communicate via coordinate variables related by formulas. We could now add a view from a new direction without going through the program and adding move-notification code to each scrollbar or scale callback. The code became easier to modify. Views were originally separate modules for each projection plane, with a large amount of code overlap. Now we have merged them into a single view module, customized by declaring appropriate formulas linking local and global coordinate variables. Altogether, by adopting the constraints model we reduced the number of lines of code by about a factor of two, and made the code a lot more understandable and extensible.

More generally, we learned two important lessons about constraints:

Constraints make code simpler. If a variable is updated in several places in the program, we only need to specify once what happens after the update. During the development of the neighborhood viewer, the neuroscientists asked to implement saving position bookmarks. When the neuroscientist

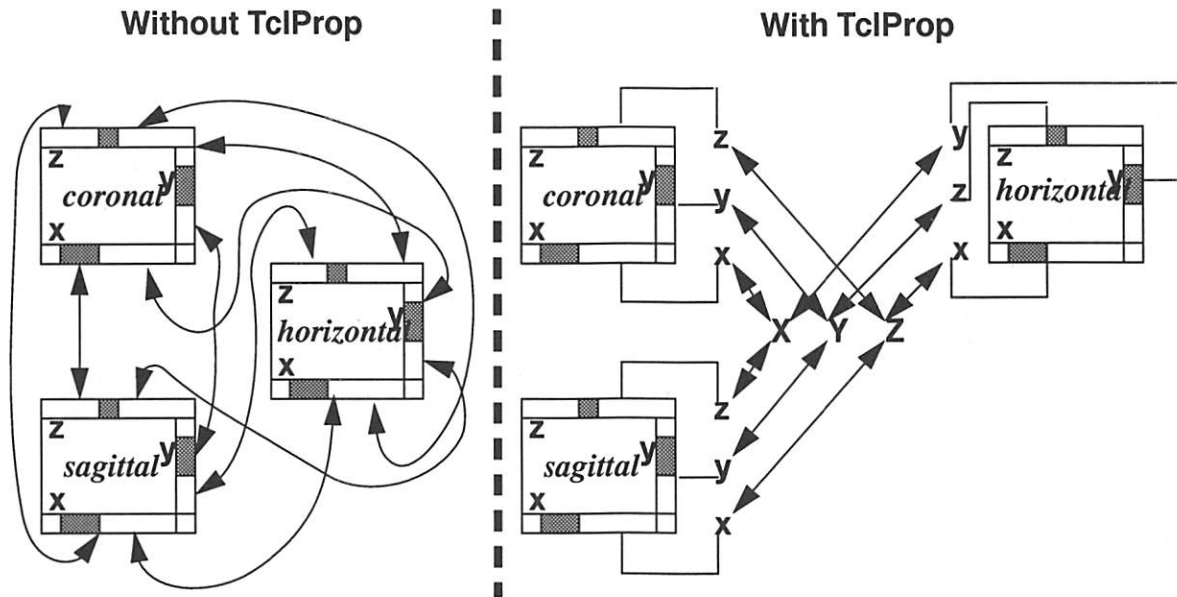


Figure 2: Two implementations of the neighborhood viewer. In the original implementation, each control widget must be aware of all other ones. In the implementation with TclProp, each control widget is linked via a constraint to a x, y, or z coordinate of the view center. The latter are in turn linked to the global position. Both links are two-way. Adding new views with TclProp requires only establishing connections between the center coordinates of the new view, and the position variable. Without TclProp, adding new views requires updating the code for each control widget of the other views.

goes to a saved bookmark, all views should center on the point with the global coordinates stored in the bookmark. If we use constraints to relate the views' centers to a global position, we just need to implement the bookmark interface and set the global position when a bookmark is selected; the constraint engine takes care of notifying the views of the new position. This localization of the update propagation code inside constraints is what makes the code simpler and more readable.

Constraints make code easier to modify and extend. Consider the task of adding a view from another stack of images. In the traditional model, we would have to add code to each existing view that notifies the new view of the change in position. With declarative programming, we create formulas that relate the view's center coordinates to the global position coordinates. Views are unaware of one another's identity, communicating only

through the constrained coordinate variables. Constraints thus allow us to focus on the part of the application being developed, while relying on the constraint engine to maintain consistency between variables.

The declarative programming model also creates new challenges to the programmer. In Tcl, constraints can be declared and executed in different contexts, so care must be taken about variable substitutions. We sometimes found it difficult to determine the program state during execution, since we had no direct control over when and how variable updates were propagated. We added extensions to TclProp to dynamically disable and enable constraints using several criteria; these extensions were useful for determining the state of the running program. We think there is a need for a "constraint visualizer" to help programmers evaluate and modify execution of a constraint-based program.

In sum, constraints are generally useful where relationships between program variables can be expressed independently of the program flow, and where there are multiple sources of variable updates. The declarative model is especially appropriate for event-driven programs, which do not have a predefined execution path, and usually have multiple ways to manipulate the program variables. The benefits of constraints are code reduction and improved extensibility, due to localized update propagation and hiding the information about dependent variables inside the constraints. Networks of constraints can be easily changed at development or runtime. We believe that the benefits of the declarative model need to be further explored by extending constraint packages, investigating domain areas that yield themselves to declarative programming, and writing more constraint-based applications.

Distributed Constraints

Imagine a group of neuroscientists in different locations, browsing through the same brain data set. As they work together, they need all their data views centered on the same position in the brain, and need to have their views updated as any of them changes. Each neuroscientist may also disconnect from the group, browse independently, then reconnect and either move to the global position, or request others to move to her position. Normally, any participant should be able to move everybody's views; however, a chairperson can be selected and made the only user allowed to control navigation. The collaborating neuroscientists thus need to control the amount and type of their interaction.

This principle, known as *flexible coupling* [4], is common to many collaborative applications. Users must be able to dynamically connect to a session, disconnect, work independently, then reconnect. GroupKit [6] defines a rich API to manage sessions, among other things. Different collaborative applications require different degrees of synchronization and consistency. We can meet these requirements with flexible update propagation: variable updates may propagate immediately, after being validated by the application, or only when the user explicitly commits them. Depending on the number of participants and the formality of collaboration, different access control methods at variable granularity are needed. Examples are floor control, locking individual variables, and locking groups of related variables.

The constraints provided by TclProp, while effective

for simple coordination of multiple linked views, are not powerful enough to implement flexible coupling of multiple users on multiple machines. To address the demands of collaboration, we needed to develop *distributed constraints*.

Distributed constraints provide a good framework for implementing flexible coupling for collaborative applications. We developed a distributed constraints module by extending TclProp to handle network communication using Tcl-DP [8], and implemented some features of flexible coupling. Dynamic disconnection and reconnection are supported by disabling and re-enabling constraints on variables of the specified view. The shared floor-holder token provides the floor control policy. Flexible update propagation is currently supported in the application, not at the toolkit level. We are currently examining ways to integrate flexible coupling into a distributed constraint module based on TclProp, and we plan to release that module at a future date.

We benefited from using distributed constraints in several ways. We were able to allow collaboration between geographically distributed neuroscientists, without significantly increasing code complexity compared to the single-user, multi-view version. As local constraints maintain views in the single-user prototype centered on the same point, distributed constraints ensure that all collaborating users see the same brain data. We achieved this by declaring distributed constraints between the coordinate positions of individual users, and the global group position. For flexibility, these constraints added a level of indirection—each user now has a “personal” coordinate that is linked by local constraints to the individual views. In this way, a user can systematically view the brain at an offset from the group view. For example, an individual neuroscientist may choose to view the brain 10 microns deeper (in the z-direction) than her colleagues, and at the same x and y coordinates. Her viewer would link to the global coordinate system as follows:

```
# link local x and y to globalX and globalY
```

```
DTP_inLink $host $port globalX x
DTP_outLink $host $port globalX x
```

```
DTP_inLink $host $port globalY y
DTP_outLink $host $port globalY y
```

```
# link local z to globalZ+10 and vice versa
```



```
DTP_inFormula $host $port \
  z globalZ {expr $globalZ + 10}
DTP_outFormula $host $port \
  globalZ z {expr $z - 10}
```

Neuroscientists can disconnect from the collaborative session and navigate through the image data independently, while keeping their personal views of data consistent with one another. This is achieved by temporarily disabling constraints between the group position and user position, while retaining the constraints between the positions of this user's views. Similarly, they can still unyoke individual views from their personal coordinate system, whether connected to or disconnected from other users. Figure 3 shows how connection and disconnection are supported.

We are experimenting with other flexible coupling models, such as floor control, within the neighborhood viewer application, and we plan to move support for these models into our Distributed Tcl-Prop module as soon as possible. We are also investigating new models including group awareness through queries into the distributed constraint table.

Incidentally, the use of distributed constraints provided our users with a valuable application feature that would have otherwise been difficult to implement. With limited screen real estate and local image processing resources, it is convenient for a single user to use several workstations and monitors, each holding a different view or set of views. Our distributed constraint approach makes implementing this feature trivial, and also provides a mechanism for distributing pieces of images that are larger than a single screen across several monitors with linked scrolling. We expect this approach both to improve performance (by parallelizing expensive image loading) and to expand the horizon of interfaces that the neuroscientists can imagine.

To summarize, we developed a distributed constraint model based on TclProp [5] and Tcl-DP [8]. Our model supports a rapid design-code-invent cycle, implements some features of flexible coupling, and supports multiple, modifiable constraint configurations. By adopting the distributed constraint model, we achieved collaboration and local speedup in the NV while keeping the code readable and extensible.

We discovered, in the process of developing and using distributed constraints, that the key to flexibility is providing places in the constraint topology

where links can be disconnected. It is important to have intermediate nodes in the graph representing users and views, to provide a site for local disconnection and transformation. Providing these intermediate nodes has not been very difficult and our observations have shown that the bottleneck of distributed constraints lies in network communication, not local constraint propagation. Therefore, any slowdown is negligible (indeed, the entire distributed constraint system performs comparably with RPC).

Finally, we believe that distributed constraints can be particularly beneficial for collaborative systems that require synchronous navigation through sets of spatial data. Examples of such application domains are medical and biological imaging, GIS, and collaborative data visualization in general.

Wavelet Lessons

Imagine a neuroscientist browsing brain data. Her two questions, "where am I?" and "what is this?", require different support. When answering the former question (or when zoomed out) not all of the detail stored in the database is needed. This situation provided us an opportunity to improve the responsiveness of our interface. We looked at reducing image transmission time by trading initial image detail for speed and, later, as it was needed or there was time, adding more detail to the image.

In order to trade off image quality for image loading speed, we needed a representation of the data that could be used to transmit an image in such a way that the quality of the image could be incrementally improved. That is, we want to incrementally improve an image's resolution, and do so at the user's command. We considered the interleaved GIF format. The popular WWW browser, Netscape, uses interleaved GIF to show the user a coarse view of a picture that is then refined with time. However, GIF was insufficient for our purposes because it can only be used to store 256-color images, and we wanted more flexibility.

Unfortunately, all of the Tk-supported image formats lacked the flexibility to store 24-bit color images in a form that can be efficiently loaded incrementally. So, we started looking at general requirements for our image model: rapid display of a good image and refinements to the image with minimal database access. If we found a suitable representation, we would see how hard it would be to extend Tk to support it.

We quickly focused on wavelet-based compressed

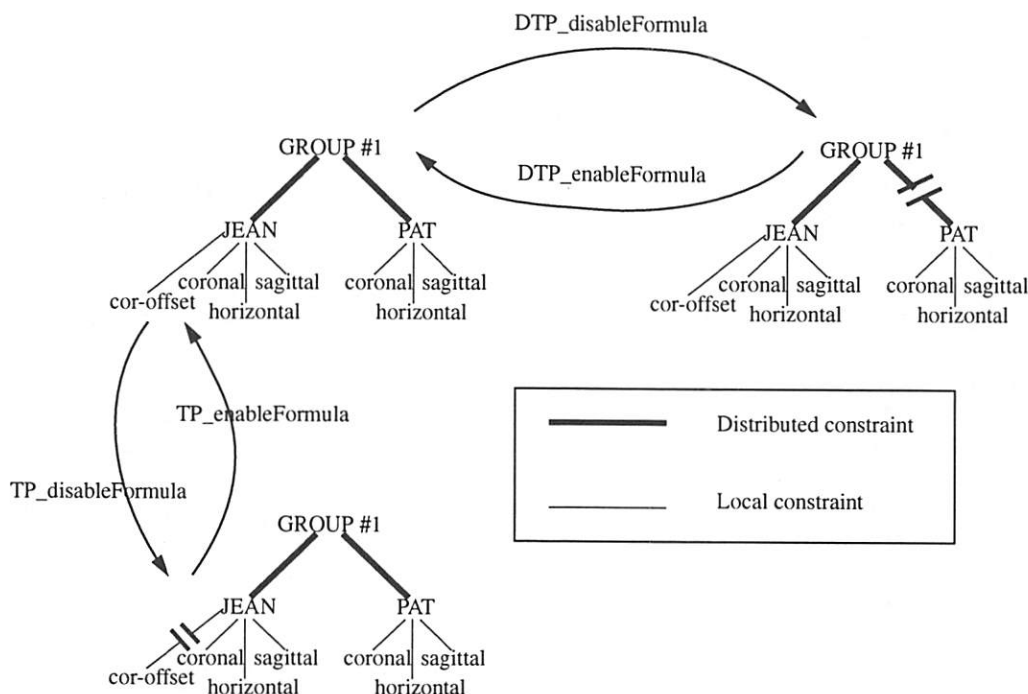


Figure 3: Flexible coupling in the neighborhood viewer. The constraints linking Jean's views with Pat's views can be disconnected at the user level (top-right) or at the view level (bottom-left).

images. For our purposes here we describe the wavelet transform simply as taking images (pixel values) and producing collections of coefficients. (Wavelets are a much more complicated topic.) These coefficients are averages and differences from the average over some area. One wavelet coefficient will represent the average intensity of the whole image whereas others contain information about smaller portions of the image. A network packet of pixels is just a set of dots, while a network packet of wavelet coefficients would generate a fuzzy but complete version of the original image.

To prepare for incremental resolution we a) transform the image to produce coefficients, b) sort the wavelets coefficients from coarsest details to finest, and c) store them. When the viewer requests an improvement in the image, more coefficients are read sequentially from disk. Sorting in this fashion satisfies our requirement that large improvements in quality occur early in the transfer. Indeed, just a few percent of the data is normally sufficient for a good image.

Wavelets work particularly well for the neighborhood viewer application. Work with early prototypes identified that users transitioned between two modes: positioning and studying. In the positioning mode, users are most interested in coarse

shapes to navigate the large number of images in the database. Once the neuroscientist has found a particular region of interest, he will study it in detail. He now focuses on a small section of the image and is interested in seeing substantial detail. Indeed, given the expense of acquiring images, he will not tolerate any loss at this level. (Another benefit of wavelet transforms is that they are not inherently lossy, which is a decisive advantage in this application over the discrete cosine transform used in JPEG.)

Our interface evolved to incorporate a new operation: add resolution. Navigation now involves a sequence of "move, add resolution" cycles as the user moves closer and closer to the point of interest.

Wavelets appear to be the right solution. They provide the model of compression we sought, one that supports rapid display of good pictures and incremental display to reach original detail. Unfortunately, wavelets involve extensive calculation, and a back-of-the envelope estimate showed that wavelet code in Tcl would never perform satisfactorily. Being ecologically-minded, we sought out a library that we could reuse, Michael Hilton's wavelet library in C, made a few modifications, and created a wavelet-compressed image object in Tcl. Our original plan was to create a new image format, but

since the current API for image formats does not support incremental image quality, we developed a "wavelet data" object that interacted with the standard photo image.

The resulting iteration of the neighborhood browser gave us a new performance trade-off. Wavelet-transformed images require less network transmission, but more computation. We find that the two versions are comparable on high-speed local networks, but that the wavelet model will become increasingly preferable as we consider transferring the images over slower, intercontinental networks. Furthermore, we are now poised to take advantage of any improvements in performance that become available for wavelet transforms (the "lifting" technique promises to double transform speed).

The lesson from this experience is that easily extendable systems make it possible to explore new technologies. Had we been developing in a system more rigid than Tcl/Tk, we probably would have abandoned the search for a better image format. Because of the clean interface for extending Tcl/Tk, we were able to integrate an existing wavelet library and explore more advanced interfaces.

We are currently working to extend our wavelet implementation in several ways. First, we are working on performance issues surrounding the inverse wavelet transform as these are the bottleneck for real-time applications. Second, we are examining techniques to handle images with widths and heights that are not powers of two. When we have addressed these issues, and defined a robust file format for wavelet data, we plan to make the wavelet extension available to others.

So, Who Gives a Rat's Brain?

While we hope the lessons we have discussed above help other developers as they build innovative applications in Tk, in many ways the more important message involves higher level principles of application design and development. In this section, we discuss some of the higher-level lessons that we learned, evaluate how well Tcl and Tk serve as development tools, and discuss some improvements.

Innovative applications do not have specifications. Developing the neighborhood browser was exciting because neither we nor our users knew what the application would become. It has been a long and continuing process for computer scientists to learn what is meaningful and useful to the neuroscientists and for neuroscientists to understand the range of options that can be provided. Most of our

progress has been made in baby steps, rather than great leaps, because great leaps require too much understanding. It is easy for a computer scientist to say "oh, we'll build you a 3D visualization of the brain," but it takes time to understand why a 3D visualization is not as useful as the right combination of 2D images. Developing innovative applications and interfaces is an inherently iterative process, and the requirements and specifications evolve as development proceeds.

Turn-around is creativity. Baby steps denote incremental rather than revolutionary progress. To invent a novel interface and application, therefore, requires many baby steps, and as a result the potential for creativity is directly related to the speed with which you can implement the next idea. It is hard to be creative when each prototype revision takes weeks or even months. With the neighborhood viewer, minor revisions could be implemented the same day or overnight and major revisions often could be implemented in only a few days. At the extreme, we would be even more creative if we could prototype revisions as quickly as we brainstorm ideas.

Your flexibility is constrained by the flexibility of your tools. If turn-around is creativity, then rapid development tools provide that creativity. It was essential to our development process to be able to change the interface and the application on short notice to obtain feedback from our users. We used Tcl and Tk because the combination of a scripting language, an interpreter that is active at run-time, and a clean interface to both stand-alone programs and compiled C and C++ code gave us the flexibility to quickly modify the interface (while an application is running) and to quickly integrate other applications and code into the application. Our rapid incorporation of wavelet-compressed images is one example of the flexibility of Tcl and Tk. At the same time, Tcl and Tk let us down in certain areas. Until we started to use TclProp, there was no flexible mechanism for connecting many views together. The difficulty in modifying change propagation stunted our ability to expand beyond simple view synchronization to more exciting and innovative interfaces including multi-screen and multi-user interfaces with flexible coupling. Even TclProp was not sufficient by itself, leading us to develop a distributed TclProp. Similarly, Tk provides few hooks to make it possible to monitor changes to widget attributes (only a few attributes have variable connections). We are examining better ways to address

this limitation for a future release of TclProp.

Nobody writes advanced programs in a low-level language. As human beings, our thinking is related to our vocabulary. Just as nobody writes a compiler by thinking exclusively about zero and one bits, we could not write the neighborhood viewer until we extended our language to include concepts such as constraints from TclProp, remote procedure calls from Tcl-DP, image handling functions from our wavelet library, and, of course, interface widgets from Tk. Extensions to Tcl are not "extras," they are the core of what makes Tcl worth using. Dynamic loading of extensions and libraries is a good step in the right direction. Still, there is great demand for further support for authors and users of extensions and libraries including: naming and version support, support for user customization, and well-indexed libraries to help people find the code they need.

Don't widen the bottom of the bottle. The bottlenecks in the neighborhood viewer are the image processing, disk, and network access. The wavelet code is all in C (quick experiments showed that Tcl was not fast enough for the computation), but all of the application assembly is in Tcl. Unless it is used for intensive computation, Tcl code is rarely going to be the bottleneck for I/O and computation-intensive applications.

Tcl/Tk is an excellent tool for developing innovative applications – let's keep it that way. We have had great success with Tcl and Tk, and are encouraged by the continuing development of new extensions and libraries that expand our vocabularies and thereby widen our horizons. Experiments with the early alpha version of SpecTcl show promise for another exciting tool that will help speed up our turn-around and increase our creativity. A key concern for the coming years is portability – the ability to distribute a Tk application without sending a new wish, and the ability to easily find and load needed extensions with version control.

Finally, there is one message that is important enough to merit its own section.

Only Dead Systems Don't Change

Each new prototype we developed engendered new interface ideas. Moreover, each prototype brought change to the way that the neuroscientists carry out research. The new interfaces are tools that let neuroscientists learn more about brains, and are not burdens. Indeed, now that browsing is easy we no longer think that the collection of images is large,

but rather are looking to fill in the gaps in the "globally sparse, locally dense database."

Depending on how you count, we produced between a few dozen and a hundred prototype versions. Had we expended any effort optimizing the code for any version we would have been wasting our time - the code was destined for an honored burial in the scrap heap. Actually such effort would have been worse than a waste, since we quite naturally would have been loath to discard what we slaved over, and would have resisted change. We did address an evident bottleneck with wavelets, but even there we just modified existing code.

Already, the fifth iteration is completing and we are heading towards a sixth, a seventh, and many many more. With each iteration, we stretch to find the tools we need to support our users. Just as the first four iterations brought us to Tcl/Tk, TclProp, distributed constraints, and wavelets; the next few iterations will lead us to discover or invent abstractions and technologies. And, with each iteration we lift our users just a bit higher, expanding their horizons, and ensuring another iteration of improvement. The story doesn't end here, it is still being written. We wouldn't have it any other way.

Acknowledgements

This research is supported by the National Science Foundation grants IRI-9410470 and IBN-9419233.

We wish to recognize the contributions to this project by our neuroscience and computer science colleagues. Thanks to Chris Honda, Tina Liang, Joe Maguire, Maureen Riedl and Sam Shuster.

References

- [1] J. Carlis, J. Riedl, A. Georgopoulos, G. Wilcox, R. Elde, J. H. Pardo, K. Ugurbil, E. Retzel, J. Maguire, B. Miller, M. Claypool, T. Brelje, and C. Honda. A zoomable DBMS for brain structure, function and behavior. In *International Conference on Applications of Databases*, June 1994.
- [2] E. Chi, P. Barry, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Visualization of biological sequence similarity search results. In *IEEE Visualization Conference*, Atlanta, November 1995.
- [3] M. Claypool, J. Riedl, G. Wilcox, R. Elde, A. Georgeopolous, J. Pardo, K. Ugurbil, J. Maguire, T. Brelje, and C. Honda. Network

requirements for 3D flying in a zoomable brain database. *IEEE JSAC*, June 1995.

- [4] P. Dewan and R. Choudhary. Flexible user interface coupling in a collaborative system. *Proceedings of the ACM CHI '91 Conference*, pages 33–45, 1991.
- [5] Sunanda Iyengar and Joseph A. Konstan. Tcl-Prop: A data-propagation formula manager for Tcl and Tk. In *Proceedings of the Tcl/Tk Workshop 95*, page 288. USENIX Assoc; Berkeley, CA, USA, 1995.
- [6] Mark Roseman and Saul Greenberg. Building Real-Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.
- [7] L. Shoop, E. Chi, J. Carlis, P. Bieganski, J. Riedl, N. Dalton, T. Newman, and E. Retzel. Implementation and testing of an automated EST processing and similarity analysis system. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995.
- [8] Brian C. Smith, Lawrence A. Rowe, and Yen Stephen C. Tcl Distributed Programming. *Proceedings of the 1993 Tcl/Tk Workshop*, 1993.

HIGH PERFORMANCE GRAPHIC DISPLAY WITH TCL/TK

Darren Spruce

European Synchrotron Radiation Facility

Grenoble, France

Email: spruce@esrf.fr

Abstract

This paper describes how Tcl/Tk was used for the development of a high speed updating graph display. Tcl is not reputed for its performance and where real time graphics is required a purpose built 'C' program is necessary to cope with the speed. However Tcl/Tk along with the BLT graph widget extension provide a useful set of tools for GUI building and graph displays. This application shows how performance can be built in to an interpreted environment so that the developer can benefit from the high level of functionality offered by Tcl/Tk and BLT and at the same time display a spectrum of 16000 points updating every second. Although the application is designed to display data from a Multi Channel Analyser (MCA), the technique can easily be adapted to display other types of 2 column data.

1 Introduction

The European Synchrotron Radiation Facility is a fundamental research institute situated in Grenoble, France. The ESRF ring is a third generation, the world's most brilliant and intense source of X-rays to date, some years ahead of equivalent facilities in the United States and Japan. When complete the ESRF will have 50 beam lines dedicated to many fields of science.

My role in the programming group of the experiments division is to make existing programs easier to use by means of graphical interfaces, to develop software for instrument control and data analysis and to research into development techniques. At present my work is mostly involving Tcl/Tk and I am also supporting a growing Tcl/Tk user community.

2 Requirements for the MCA graph display

The Multi Channel Analyser is a device which counts electronic pulses and groups them into channels according to their height. It is typically attached to a

detector and configured so that each channel corresponds to a different energy or time. Since the MCA only knows about channel numbers and counts, the channels need to be calibrated. The program needs to provide the possibility for the user to enter the calibration. The output is best displayed as a graph of desired units (energy/time) versus counts, and this graph is changing in real time as the counts increase for each channel. Fast visual feedback during acquisition of data permits the user to make adjustments (for example during setup of the ADC).

Overall the application has to be easily adaptable to the varying requirements according to its particular use - some parameters are necessary for some users and unnecessary for others.

3 Choosing the Software

At the ESRF we have well established software for data acquisition. The problem was to redefine the user interface in a way to enable him to see and manipulate the data interactively as well as control the hardware.

What was needed was an independent application that could look at the data and present it in a user friendly and interactive way. It was necessary to look for high level tools to speed development and reduce the amount of work needed to complete the task because of the limited resources.

There were two suitable toolkits available, one was a commercial graph toolkit which could be used with a user interface builder, the other was Tcl/Tk with the BLT graph extension. The commercial graph widget required that a license fee should be paid for each piece of executable code that used it and since the ESRF is a public user facility where it is desirable to freely distribute software this is not very convenient. After comparing the functionality of both toolkits and seeing that they were similar I decided to first test the BLT toolkit for its performance.

The BLT graph widget proved to have lots of functionality but was not fast enough to cope with the amount of points and the refresh rate. So I resorted to using a 'C' routine which charged the BLT graph with the data. This was not difficult using the 'standard' methods to extend the Tcl commands and resulted in a much better performance.

The solution seemed to be to use the tools provided by Tcl/Tk and BLT to build a graphical interface and write some underlying 'C' code to handle the data. The transfer mechanism between the graphical interface program and the underlying software was implemented with shared memory. It is ideal for large amounts of data where synchronization is not too important and is compatible across the UNIX platforms used at the ESRF.

4 Design

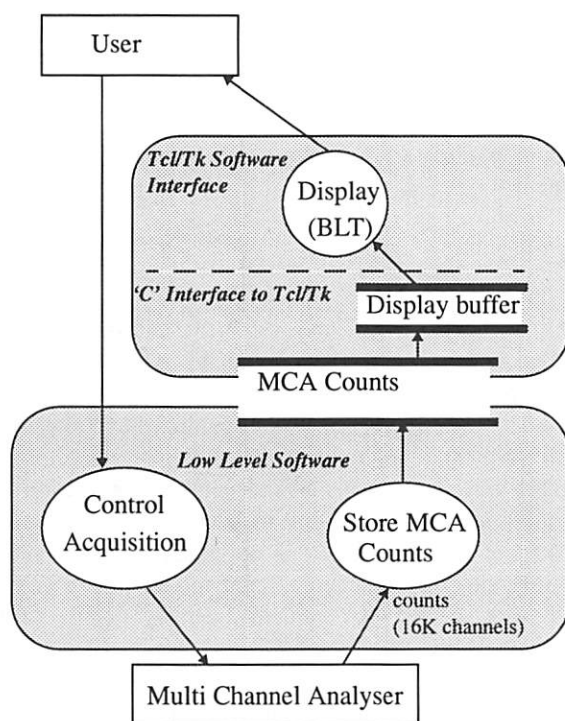


Fig 1. Data Flow Diagram showing User Interface to the MCA.

The data flow diagram above (Fig 1) shows the complete system to control the MCA and display the data. The object bridging the gap between the Tcl/Tk interface and the low level software is the shared memory area used to store/read the data. This is the only

place where there is any dependency on the format of the information with the lower level software/ hardware.

The underlying software takes care of the details of talking to the equipment, the instrument control and the creation of the shared memory from which the MCA GUI collects its data. The data structure is the only constraint which ties the 2 pieces of software together. The structure used allows the update to be synchronised by the arrival of new data, and permits identification using a 'magic number'.

Although the BLT graph could handle the refresh rate of the data it was inefficient in that it attempted to display all the points even if they were invisible due to the screen resolution. This loaded the X server heavily inhibiting other applications from running. However after a few days of further 'C' programming I improved the data formatting routine so that it reduced the raw data to the same equivalent number of points as the pixel width of the graph display. This changed dynamically as each call to update the graph included the current width of the graph. The modification successfully reduced the load on the X server.

5 Key Implementation Details

The following data structure defines the link between the underlying software and the display application.

```

struct shm_header {
    long magic; /* magic identifier */
    long type; /* data type */
    long version; /* of shared memory */
    long rows; /* number of rows */
    long cols; /* number of columns */
    long utime; /* last updated time */
    char name[NAME_LENGTH];
    char spec_version[NAME_LENGTH];
    char pad[SHM_HEAD_SIZE (6*
sizeof(long) +2*NAME_LENGTH)];
    void *data;
};

```

This structure originated from the underlying software and is used to implement the shared memory extended commands in the interpreter's display.

The Tcl Interpreter is extended with a number of commands. Some are for analysis such as peaksearch and polynomial fitting, and others for calibrating the data. These are not covered in this paper.

The key Tcl extended commands are:

get_shmid

This command reads the list of shared memory segments on the system and returns the id, name, version, number of rows and columns of the segment if it matches the pre-defined magic number. This permits the Tcl application to recognise and select between segments.

read_shmid shared_id force_read

Given the shared memory id and a flag this command checks to see if the memory has been updated by looking to see if the utime variable has changed.

replot pathname min xmax width mode

Takes the arguments, pathname of graph, minimum X value, maximum X value, the pixel width of the graph and the mode. This command gives big performance advantages over using Tcl to plot the data. It takes the currently selected buffer of raw data (there can be more than one) and reduces the data to the specified pixel width. It is also necessary to format and convert the type to make it suitable for BLT display call. The mode gives the possibility for the replot function to use a calibration function when converting the raw data.

The program execution starts by creating a window with a graph display and then initiates a poller which looks at the shared memory on the system. The poller code is written in Tcl with the extended command 'get_shmid' to read the shared memory. The name of the shared memory can be specified at runtime, so that if more than one application is running they can look at different areas of shared memory.

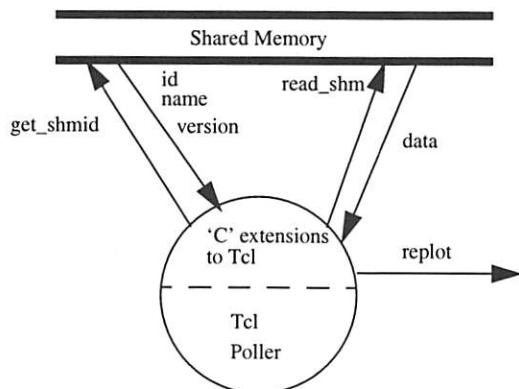


Fig 2. Poller interaction with shared memory

The poller runs slowly (once per second) unless it finds a matching identity. This gives the application more time to respond to user events when there is no shared memory to be updated. When an area of memory is found that matches the input arguments of the program the poller increases speed to approximately 10 times per second, now calling `read_shm` to read it if it has changed. If the contents have changed then they are read and the next 'test if changed' is scheduled in 0.3 seconds. This means the memory will be read at a maximum rate of 3 times per second if it is always updating and the delay before an update is about 0.1 seconds. The data is read into an intermediate buffer in raw form before it is formatted for the BLT graph.

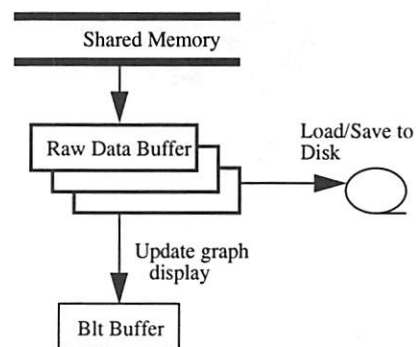


Fig 3. Buffers used by the application

There are 2 buffers in the program, one for the raw data and the other for the graph display. Each time a change is made which requires a re-plot of the display (e.g. window resize, zoom in/out or scroll) the `replot` command is called which supplies the graph width, range and calibration mode to the plot routine. This routine formats the relevant area of raw data and then copies it into the graph widget buffer. This is necessary to be able to save or load the raw data since not all of it is contained in the Blt buffer. It also allows multiple plots in the same window.

Every time the graph display is updated either by resizing the window, zooming or scrolling the ratio of pixel points against data points over the zoomed region is re-calculated and the corresponding number of points read from the memory buffer to the blt buffer.

6 Program Functionality

This section describes the various functions of the program. It is included to give a flavour of the high level

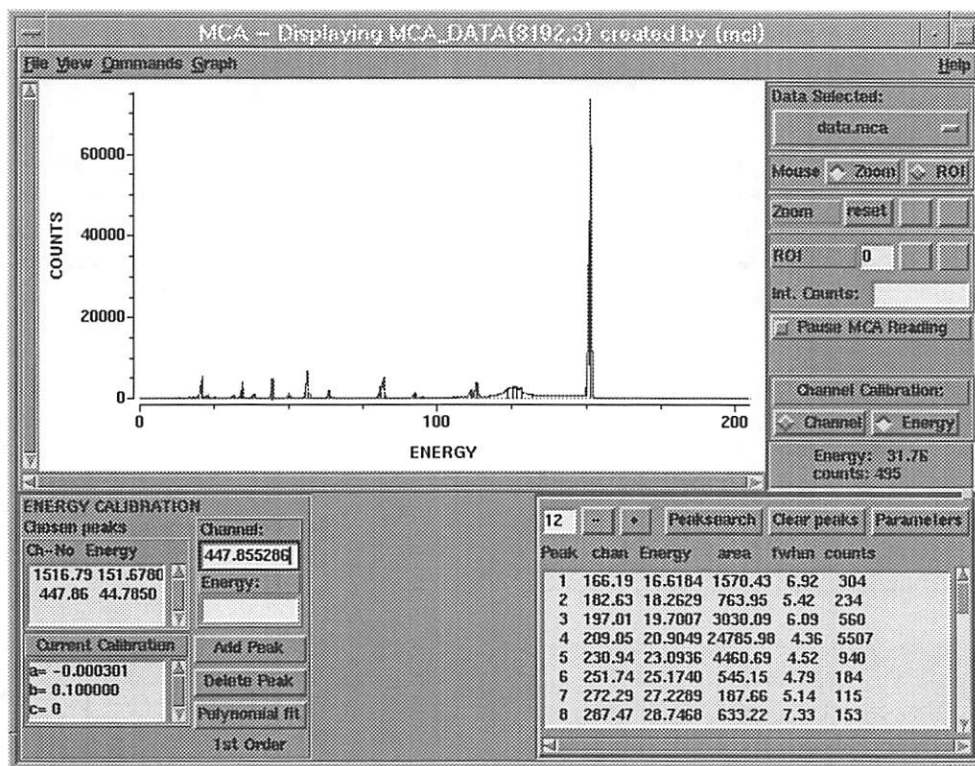


Fig 4. Overall appearance of the Window

of functionality achieved by using the Tcl/Tk/Blt/Tix toolkits.

6.3 Graph control

6.1 Window creation

The main window was originally built using only Tcl/Tk and BLT. It was later modified to make use of the advanced features of the Tix widget toolkit such as the file selection box and paned windows. The other main benefit of using Tix was the very neat default colours and appearance.

6.2 Update Graph Poller

See 'Key Implementation Details' for information on the poller.

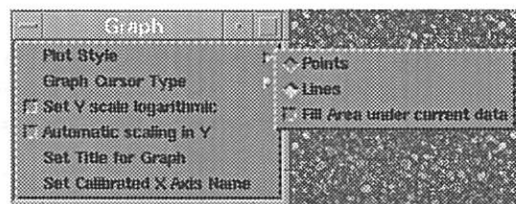
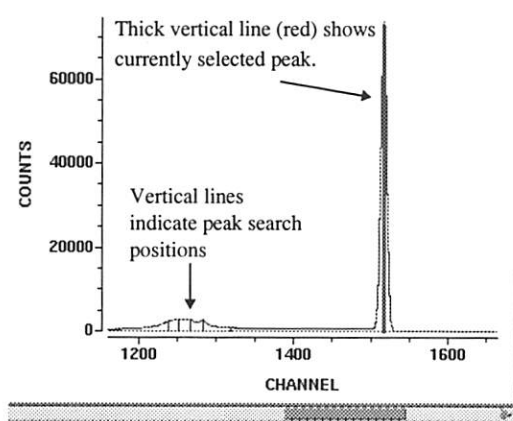


Fig 5. Graph Style Menu

Following an action by the user the graph often needs updating. The Tcl code handles the parameters of replotting and then calls a 'C' routine to update the plot. The display styles of the plot are also controlled with Tcl. Originally I used a demonstration program supplied with BLT to control all its resources and this provided a good way to experiment with the graph appearance. Finally it was necessary to limit this to a more manageable set of display options.

6.4 Peak Search

The user can start a peak search. This will look for peaks in the currently selected plot. A 'C' routine looks for peaks, and the positions are returned to Tcl. The Tcl code handles user selection and displaying of the peaks and also a way to enter them into the calibration.



Peak	chan	Energy	area	lwfm	counts
26	1134.26	30715.33	8.14	3844	
27	1237.61	18036.07	14.74	2293	
28	1251.82	26905.11	16.98	2682	
29	1266.27	24419.44	14.95	2763	
30	1282.74	17576.06	10.77	2586	
31	1319.30	385.33	4.14	1010	
32	1516.79	675474.69	8.64	72291	
33	1712.40	203.71	16.50	14	

Fig 6. Peak Search facilities

6.5 Regions of Interest

It is possible to define areas of the graph marked by vertical lines defined by the user with the mouse. When selected (shaded in grey) any peak search is only done over the range of the shared region. The integrated counts over that region are also calculated and displayed. The regions of interest are definable by

dragging with the mouse or by entering the boundaries using entry boxes.

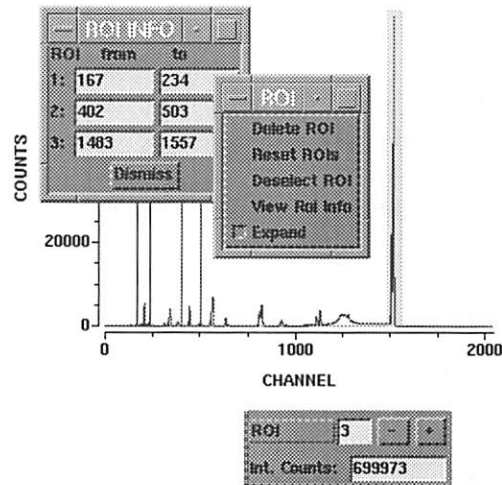


Fig 7. Selection of windows for ROI selection.

6.6 Zoom Facilities

Zooming can be performed either with the mouse by clicking and dragging over a region or by entering the desired boundaries of the zoom region. Each zoom is stored in a list which can be reset at any time. There is also an expand function for regions of interest where a zoom is performed to the limits of the currently selected region of interest. A zoom can be performed while the display is changing and so it is possible to choose whether the graph continues to do automatic scaling in the y axis or to freeze the zoomed region.

6.7 Scrolling

Each time the user zooms in it is still possible to scroll over the whole range of the data. The graph updating is handled by 'C' code because the BLT graph does not contain all of the data. When scrolling the calculations for the scroll bar sizes and the scroll step size is done with tcl and the 'C' function to update the graph display is called.

7 Performance

The running program consumes between 7% and 15% of the CPU of an HP9000/715 machine along with a similar load on the X server. This leaves enough resources of the machine for the underlying application which reads the MCA data and for other beam line applications. One interesting feature is that the CPU

usage of the program and the X server is proportional to the window size as the amount of work to be done to refresh the graph is related to on the number of pixel points to be plotted. If the user is less interested in the display he can reduce the window size therefore reducing the CPU load. Only when he becomes more interested in the display does he enlarge the window and increase the CPU load. The screen refresh rate is at about 2 or 3 times per second depending on the CPU load and if you are using an X terminal, the amount of network traffic.

8 Conclusion

Tcl/Tk, BLT and a later addition the TIX extension proved to be robust software toolkits which were easy to build on, providing a high amount of functionality coupled with easy extendability. This example has showed how the best of both worlds (performance and high level programming) can be achieved by using an interpreted language.

9 References

- [1] John K. Ousterhout: An Introduction to Tcl and Tk
Computer Science Division. Department of
Electrical Engineering and Computer Sciences.
University of California, Berkeley. Draft for
Addison Wesley 1993
- [2] George Howlett: Blt 1.7, Blt1.8.
gah@grenache
- [3] Ioi Lam, Tix 4.0b8 1995
ioi@CS.Cornell.EDU
- [4] Brent B. Welch:
'Practical Programming in Tcl and Tk',
Prentice Hall PTR 1995

Hypertools in image and volume visualization

Pierre-Louis Bossart

Lawrence Livermore National Laboratory

Non-Destructive Evaluation Section

L-416, 7000 East Avenue

Livermore, CA 94550, USA

Phone: 510-423-9350 Fax: 510-422-7819 E-mail: bossart@redhook.llnl.gov

Abstract

This paper describes our experience in image and volume visualization at the Lawrence Livermore National Laboratory. After an introduction on visualization issues, we present a new software approach to the analysis and visualization of images and volumes. The efficiency of the visualization process is improved by letting the user combine small and reusable applications by the means of a machine-independent interpreted language such as Tcl/Tk. These *hypertools* can communicate with each other over a network, which has a direct impact on the design of graphical interfaces. We first describe the implementation of a flexible gray-scale image widget that can handle large data sets, provides complete control of the color palette and allows for manual and semi-interactive segmentation. This visualization tool can be embedded in a data-flow image processing environment to assess the quality of acquisition, preprocessing and filtering of raw data. This approach combines the simplicity of visual programming with the power of a high-level interpreted language. We show how hypertools can be used in surface and volume rendering and how they increase the interaction efficiency by performing complex or tedious tasks automatically. One biomedical application is presented.

1 Introduction

Visualizing images or volumes helps extract qualitative information and quantitative measurements from raw data sets. Visualization software is thus becoming critical in virtually every domain of engineering. However, despite the number of packages available from commercial vendors and from the public domain, it is extremely difficult to find

a package that fulfills the needs of a research laboratory dealing with large data sets. Indeed, all the packages are implemented based on the following scheme.

- First, the file formats need to be decoded, and the raw data read in. The data can then be formatted or extracted. This step includes operation such as subsampling, interpolation or dimensional reduction, e.g. when a slice is extracted from a 3D volume.
- Next, the data are normalized, typically to 8- or 16-bit integers, and displayed in a window after the colors have been allocated. Using predefined color look-up tables, stretching the histogram, reducing the image dynamic range and providing a colorbar help identify the relevant features of the data set.
- Numerical values, extracted data or regions of interest can then be written back into a file.
- At any point, the user may interact with a graphical user interface (GUI) or issues commands to an interpreter, e.g. to change the color look-up table, look at pixel values, etc.

Practical experiments showed that none of the all command-line oriented (VIEW, IDL, PWave, Matlab, etc), dataflow-oriented (AVS, Explorer,...) or "self-contained" (Analyze, 3DVIEWNIX, etc) visualization packages could handle large data sets whose size exceed both the size of the memory and swap space. The user is thus compelled to manually extract smaller pieces of the data sets, which can be time-consuming and inefficient. It is thus mandatory to tightly link the data extraction and data visualization steps, typically by reading and visualizing one slice at a time instead of loading a 3D volume in memory before visualizing its 2D slices.

The efficiency of interactive visualization is also limited by the GUI design, which can almost never be customized by the user. Even when the source code is available, adding or removing features is very difficult if not impossible, as the GUIs are implemented as monoliths of hundreds of thousands of lines. However, the user needs to control the way the colors are allocated, e.g., interactive thresholds, linear and non-linear colormaps. Similarly, extracting profiles, histograms or non-rectangular regions of interest make quantitative measurements possible. Off-line data transforms or extractions are in our opinion too cumbersome in a research environment. In addition, unsupervised automatic segmentation performs poorly when the data are noisy. The alternative, that is the use of manual and semi-automatic segmentation techniques, is however limited by the lack of flexibility of most graphical interfaces.

The considerations above led us to implement a new visualization tool, geared to large data sets. Since we could not afford, nor had the experience required to write a self-contained application in X-Motif, our approach was to divide the GUI into small reusable components by relying on Tcl/Tk. Tcl/Tk is now used by thousands of users in every domain of graphical and engineering applications. Indeed, Tcl/Tk provides simple ways to "glue" different modules together and it can be extended easily, in contrast to other GUI builders.

However, the use of Tcl/Tk in the signal and image processing community is scarce. Two main reasons can explain this situation. First, efficient data management mechanisms and number-crunching capabilities in image processing are generally believed to require high-performance languages, in contrast to Tcl which only handles character strings. Next, the photo image widget was not designed for interactive visualization and its flexibility is very limited, mainly because image processing applications are not the main focus of Tcl/Tk developers.

In this paper, we describe the implementation of a new gray-scale image widget. By focusing on data management and color allocation problems, we were able reach a level of performance which compares favorably with other image processing and visualization packages. A transparent overlay mechanism provides a link to manual and semi-interactive segmentation techniques. We will show how this widget helped us build VISU, a flexible VISUalization software described below. VISU is made up of several standalone applications which communicate with each other over the network. The power and flexibility of these *hypertools* in image and volume visualization will be described with some examples. A

biomedical application will be presented.

2 The *pict* gray-scale image widget

As it can be guessed from the name, the *pict* gray-scale image widget is based on the *photo* image widget. After few experiments, it became clear that the *photo* widget was not appropriate for our application. First, this widget can only handle 8- or 24-bit color images, and it does not provide any mechanism to visualize floating-point images. Next, the colors are allocated statically and cannot be changed dynamically by the user. Thus, we decided to implement a new widget by focusing on data management and color allocation.

2.1 Data management

Since we wanted to support byte, short, integer or float-point types, we modified the data structure, in order to remove the fields related to color management and dithering. The raw data is allocated in a block of memory and can be accessed by using the data type information. The master structure also provides a pointer to a block of byte data, corresponding to the normalized raw data. The contrast can be increased or decreased by setting the dynamic range of the raw data. This feature proves most helpful when comparing two floating-point images whose dynamic range is different. The pixel values can be queried, and the result string contains the actual value, for example a floating-point value.

The Khoros1 and VIEW (local LLNL format) file formats are now supported. The GIF and PPM readers were modified in order to read only one color band. Readers/writers for the SUN Raster file format are provided for both the photo and the pict widget. Raw binary files can also be read from a file (or a channel if the code is linked against Tcl7.5) by specifying the dimensions, the data type and the number of bytes corresponding to the header. In all cases, only one slice is read at a time, which decreases the memory requirements dramatically. However, volume rotations and transpositions need to be done off-line. Support will be added in the near future for HDF, netCDF and ACR-NEMA file formats.

Profiling the Photo source code showed that memory management was fairly inefficient, as a lot of time is spent copying blocks between different addresses. The TkPictPutBlock and TkPictPutZoo-

medBlock routines were rewritten to make sure memory blocks are duplicated only when necessary.

2.2 Color allocation

The human visual system cannot see more than 60 shades of gray, which makes color management for gray-scale images much more simple than for color ones. Since we wanted to change the colors dynamically and use predefined colormaps, we chose to display the images using an 8-bit PseudoColor Display. This requirement is in our opinion fairly minimal. Besides, our experience proved that dithered images cannot be compared accurately.

In order to allow for fast array transformations, the colors are allocated from a contiguous set. For example, the palette can be inverted quickly by reversing the color indices. The images are displayed in false colors by choosing from a variety of predefined look-up tables. The histogram can be stretched or thresholded to produce a binary image. Furthermore, the colors can be allocated from shared, default or private colormaps. Changing the colors of one shared colormap will affect all the images that share it. This feature allows the user to visualize the same image displayed with different colors, or to compare the result of two different thresholds.

One additional benefit is that the color allocation can be used to display "semi-transparent" overlays. This is a feature that was found very useful in our interactive segmentation work. The user can for example "paint" on the image, draw polygons, Bezier or free-form curves, and yet guess the gray-level values. The overlays can be saved as a mask image. Let us point out that this feature enables the user to extract non-rectangular regions of interest. Alternatively, a mask image can be overlaid on top on the active image, in order to check an off-line segmentation over, remove spurious pixels or compare two data sets. The user can interactively combine overlays with logical operations (or, xor, and, etc..) by changing the Graphic Context. The *pict* widget provides an interface to advanced image processing routines [1, 2], and the overlays can also be used in semi-interactive segmentation [3], where a coarse initialization is specified; the segmented result can then be displayed. Examples in Figure 1(a)-(d) show two cross-sections of a CT volume displayed with different look-up tables. The concept of transparent overlays is described in Figure 2(a)-(c). The use of overlays in semi-automatic segmentation is presented in Figure 2(d)-(f).

3 VISU: a volume VISUalization application

In this section, we describe how the *pict* widget is incorporated in VISU, a user-friendly volume visualization extension of *wish*. In addition to the low-level widget commands, we provide a set of default scripts and high-level commands which make the life of the average user easier. In order to remove any learning curve for LLNL users, the syntax of these commands reproduces that of VIEW, a general signal processing package used in our group. For example, the following commands will read the first slice of volume "hand", display the image, and then display the tenth slice.

```
% rdfile hand.0.sdt f
% disp f
% rdslice f 10
```

The simplicity of these commands enables the user to write his/her own set of macros. In addition to the command line, VISU also provides a graphical interface to most of the high-level commands. In order to reduce the GUI design, we chose to provide only one Control Panel (see Figure 3) and a one-to-one mapping between images and windows. At a given time, one image is considered "active". An image becomes active when the user clicks on it, and its window title is changed. The mouse location and pixel values displayed in the Control Panel correspond to those of the active window. Similarly, moving the slice scale will result in another slice being displayed; the dynamic range of the images can be typed in two entry boxes.

The Palette Menu (Figure 4) lets the user change the colors with the mouse. Four scales can be moved in order to choose the low and high thresholds. The pixels whose values are between the low and high threshold appear white, and the rest appear black. As soon as the scale is changed, the look-up table is updated. Visualizing the actual values of the thresholds instead of normalized values proved very helpful. For example, in our CT applications, the user can see where the attenuation value exceeds a threshold. Predefined look-up tables can be loaded by clicking on one of the radiobuttons. A color tool makes it possible to stretch the colormap in a non-linear way. The graphical interface allows the user to change the intensity and each of the RGB channels independently to create their own look-up table.

In the Overlays Menu (Figure 5), the active mask can be chosen and overlaid onto the active image. The user can choose how to combine the overlays by

setting the overlay Graphic Context with the mouse. For example, the intersection of two binary masks can be seen and saved into a new image. We also provide a graphical interface to our segmentation routines.

The GUI can be customized within minutes without recompiling any code. For example, displaying several images side-by-side could be done by packing them in the same canvas, instead of different windows. This flexibility enables us to design the best GUI and to take into account the requirement of a specific imaging application. In most cases, the user will be able to configure the GUI himself.

Releasing the source code on the Internet¹ helped test VISU on various platforms and operating systems we did not have access to. The *configure* tool generates Makefiles automatically. While it is still dependent on Xlib, VISU can be ported to Windows and Mac-OS without too much effort, to become a machine-independent visualization software. About 500 anonymous ftps were logged on our server. Let us remark that this figure may appear small, but most of the VISU users have very specific needs and would probably not use Tcl/Tk at all if this gray-scale image widget did not exist.

4 Hypertools

So far, the features of VISU we described are fairly standard, and can be found in other less flexible visualization packages. However, the Tk library makes it possible to use our widget to build visualization *hypertools*, defined in John Ousterhout's book [4] as "stand-alone applications which can communicate with each other and be reused in ways not foreseen by their original designers". Indeed, it came as a surprise to us how easily the *send* command can be used by visualization tools based on the *pict* widget in peer-to-peer or master/slave relationships. In order to demonstrate the power of these hypertools, we take several examples.

4.1 Image visualization

We mentioned several times in this paper the importance of normalizing floating-point images in the same way. Typically, this can be done by querying the minimum and maximum value of each image, and by computing the absolute minimum and maximum of all the images being displayed. Setting the dynamic range has to be done each time an image is updated; for example, in volume visualization,

the dynamic range is likely to be different for each slice. Moreover, visualization tools may run on different systems and display the images on the same screen. In summary, choosing a correct dynamic range is a tedious time-consuming task. However, the *send* command makes things simple, as the following script demonstrates:

```
proc set_range {} {
    # initializations
    set l [wininfo interps]
    set min {}; set max {}; set visu_list {}

    # list visu applications
    foreach k $l {
        if { [string match "visu*" \
            [lindex $k 0] ] } {
            lappend visu_list $k
        }
    }

    # query dynamic range of active images
    foreach k $visu_list {
        lappend min \
            [send $k {$curr_img getmin}]
        lappend max \
            [send $k {$curr_img getmax}]
    }

    # compute the global minimum and maximum
    set fmin [lindex $min 0]
    set fmax [lindex $max 0]
    foreach k $min {
        if { $k < $fmin } {
            set fmin $k
        }
        if { $k > $fmax } {
            set fmax $k
        }
    }

    # set the new dynamic range
    foreach k $visu_list {
        send $k {$curr_img range} $fmin $fmax
    }
}
```

In this script, the dynamic range is broadcast to all the visu applications. Other possibilities include showing the pixel values at the same mouse location in different images, so as to compare images on a pixel-by-pixel basis. Using the visu scripts, this option could be written as the following command.

¹ftp://redhook.llnl.gov/pub/visu

```
foreach k $visu_list {
    send $k {set x $x}
    send $k {set y $y}
    send $k {set pix [$curr_img get $x $y]}
}
```

Instead of sending values, it is possible to send commands, and for example to visualize the same slice in different data sets by broadcasting the command `rdslice $curr_img $curr_slice`. A direct application is the creation of simultaneous animations of volumes located in different file systems and accessed over the network.

4.2 Image profiles and histograms

Visualization of image profiles along an arbitrary line is an invaluable tool to evaluate the presence of artifacts in reconstructed images. For example, in X-ray CT, beam-hardening produces “cupping” artifacts that can easily be detected by extracting a profile. Similarly, the accuracy of the reconstruction techniques can be assessed by visualizing the profiles of sharp edges. Typically, low-pass filtering and noise elimination smooth and spread the transitions.

Along the same lines, visualizing a histogram helps understand the statistics of an image and the distribution of its gray levels. In the case where the histogram is made up of several modes, the objects can typically be segmented out by applying different thresholdings to the data.

Although most image visualization packages provide 1D signal viewers, our experience proved that they cannot be easily customized by the user and are not flexible enough.

In our applications, the 1D signals are extracted by issuing a command to the *pict* image widget and sent to a BLT graph. Tcl/Tk handles only character strings and relies on the X protocol to communicate between applications. In addition, the profiles are generally made of less than 1000 samples, so that the overhead introduced by the communications is minimal. This feature allows us to let the user configure the graphical interface. Figure 6 shows an example where a profile is extracted interactively and then sent to a 1D signal viewer. The main benefit of this approach is that the user can dynamically create and extract new profiles or histograms of images viewed across the network, combine and compare them without having to save these one-dimensional signals in files, and transfer them by ftp.

4.3 Data-flow environments

In our biomedical application, all the data preprocessing and segmentation is implemented in Khoros, a data-flow image processing environment. Khoros provides a network editor; the input data undergo a series of transformations each time a network node is executed by the scheduler (Figure 7). This data flow environment helps the user design an imaging application, as the output of each node can be visualized by connecting it to a visualization module (other packages such as AVS, Explorer, LabView use the same paradigm). However, dynamically setting the range of all the images in a data flow environment is next to impossible. Similarly, writing a macro or a loop in a command-line interface sometimes makes more sense than editing a network of nodes.

These two problems were solved by implementing a Khoros module that executes a VISU script. This approach results in several important benefits. First, it compensates for some of the drawbacks of VISU, e.g. by providing simple ways to extract arbitrary slices. Basically, we rely on modules provided in the Khoros distribution, or write our own modules to filter and transform the raw data. Next, the power of both command-line and data-flow environment can be combined: a command-line interface is provided by a remote controller derived from the *rmt* example of the Tk distribution, which can communicate either with a specific application or send messages to all of them. As a result, the user can rely on the power of visual programming, while being able to rely on a more traditional command-line interface. To our knowledge, this feature is not provided by any other visualization package. Finally, the embedded visualization modules can communicate with each other. For example, the same profile could be extracted in different images and displayed in the same 1D signal viewer. Let us remark that the same approach could be easily implemented in other data-flow environments such as AVS or Explorer.

4.4 Volume visualization

As described before, VISU can display slices of a 3D volume. Another way to visualize a volume is to extract isosurfaces, either by using the 3D Marching Cubes algorithm [5] or by reconstruction of a 3D surface from 2D contours [6]. Visualizing a surface helps understand the spatial distribution of objects in 3-space, in contrast to visualizing 2D slices. Almost every visualization package provides these tools, but they are almost never combined or used simultaneously.

Several other approaches make use of Tcl/Tk in volume visualization. Schroeder, Lorensen and Martin recently released vtk², a visualization toolkit which can be used either by writing C++ programs, or through Tcl/Tk scripts. Similarly, Lacroute [7] released VolPack³, a volume rendering library coupled with a Tcl/Tk-based graphical interface.

These two toolkits could be used in conjunction with VISU in order to build a volume visualization package by relying on the `send` command. This approach results in two main benefits.

First, the different visualization modules can be combined easily, even if they are not executed on the same machine. Typically, it will be possible to highlight a point of the surface and to see the intersecting slice. Conversely, viewing a new slice will automatically change the coordinates of this highlighted point. The color tools provided with VISU will also be used in order to choose isosurface values, or the transfer functions in volume rendering. The same graphical interface can also be used to set the viewing parameters in surface and volume rendering. It is our belief that the combination of these visualization techniques will help identify more efficiently the relevant features of volumetric data sets.

Next, provided that the interface between hypertools does not change, the user does not need to know how the surface or volume rendering techniques are implemented. As a result, the best visualization modules can be chosen by the user. For example, vtk supports an abstract rendering engine and can be slower than the tkSM widget⁴, which provides easy access to the OpenGL and Mesa libraries. In the case where the software modules were not linked against Tcl/Tk, they can still be used and executed from the Tcl/Tk environment.

5 Conclusion

Although Tcl/Tk is widely distributed in engineering and graphical applications, its use in imaging and visualization is recent. In this paper, we described a new gray-scale widget and its use in visualization of large data sets. Although it relies on an interpreter, its speed compares favorably with typical X-Motif self-contained applications, as we implemented an efficient management of data and colors. Designing high level commands and combining hypertools helps perform tedious tasks automatically, thus increasing the interaction efficiency. We plan

in the near-future to enhance our image processing library and to port the code to Windows and Mac. The Tcl7.5 sockets will also be used to develop distributed visualization applications for the Internet.

6 Acknowledgments

This research was performed under the auspices of the U.S. Department of Energy, contract No. W-7405-ENG-48 and LDRD grant No. 96-ERI-003, with the support of H. Martz and K. Hollerbach. J. Pearlman, M. Abramowitz, E. Nicolas, J.P. Hebert, M. Cody and D. Garrett helped improve VISU by providing invaluable bug reports and suggestions.

References

- [1] L. Vincent and P. Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *I.E.E.E. Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583-598, June 1991.
- [2] L. Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *I.E.E.E. Transactions on Image Processing*, 2(2):176-201, April 1993.
- [3] P-L. Bossart. Détection de contours réguliers dans des images bruitées et texturées : association des contours actifs et d'une approche multiéchelle. Thèse, Institut National Polytechnique de Grenoble, Octobre 1994.
- [4] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994. ISBN 0-201-63337-X.
- [5] W.E. Lorensen and H.E. Cline. Marching Cubes: a high resolution surface extraction algorithm. *Computer Graphics*, 21(3):163-169, 1987.
- [6] B. Geiger. *Three-Dimensional Modeling of Human Organs and its Application to Diagnosis and Surgical Planning*. PhD thesis, Ecole des Mines de Paris, 1993.
- [7] P.G. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University, 1995. also Technical Report: CSL-TR-95-678.

²<http://www.cs.rpi.edu:80/~martink>

³<http://www-graphics.stanford.edu/software/volpack/>

⁴<http://www.isr.umd.edu/ihsu/tksm.html>

7 Figures

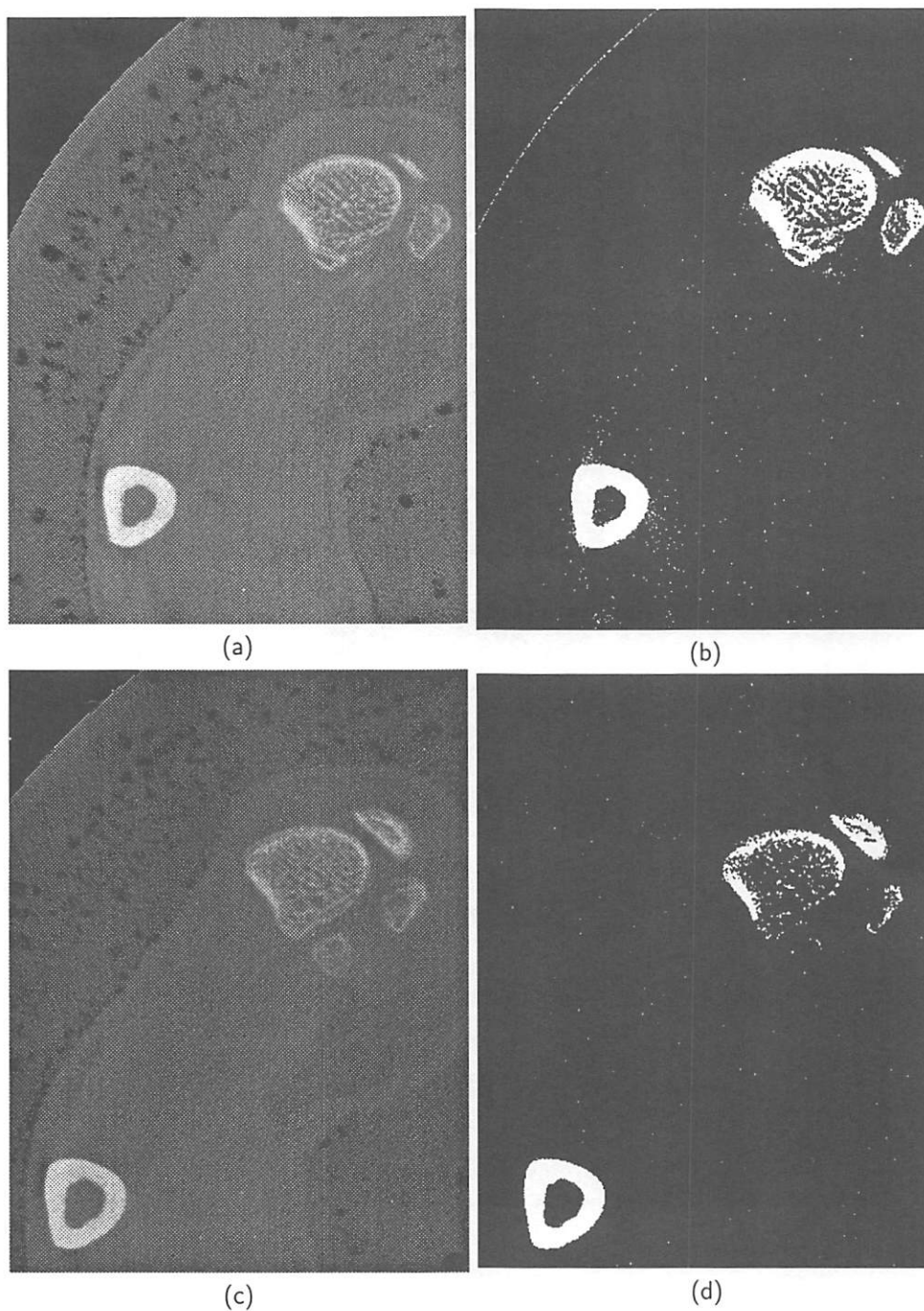


Figure 1: Cross-sections of a thumb and index near joint: (a) corresponds to slice 308 displayed with a 'ct' private colormap (b) is also slice 308 with a shared colormap (c) corresponds to slice 314 with a 'gray' private colormap (d) is slice 314 with a shared colormap. In this example, the private colormaps are used as a reference while the colors are changed simultaneously in the two shared colormaps.

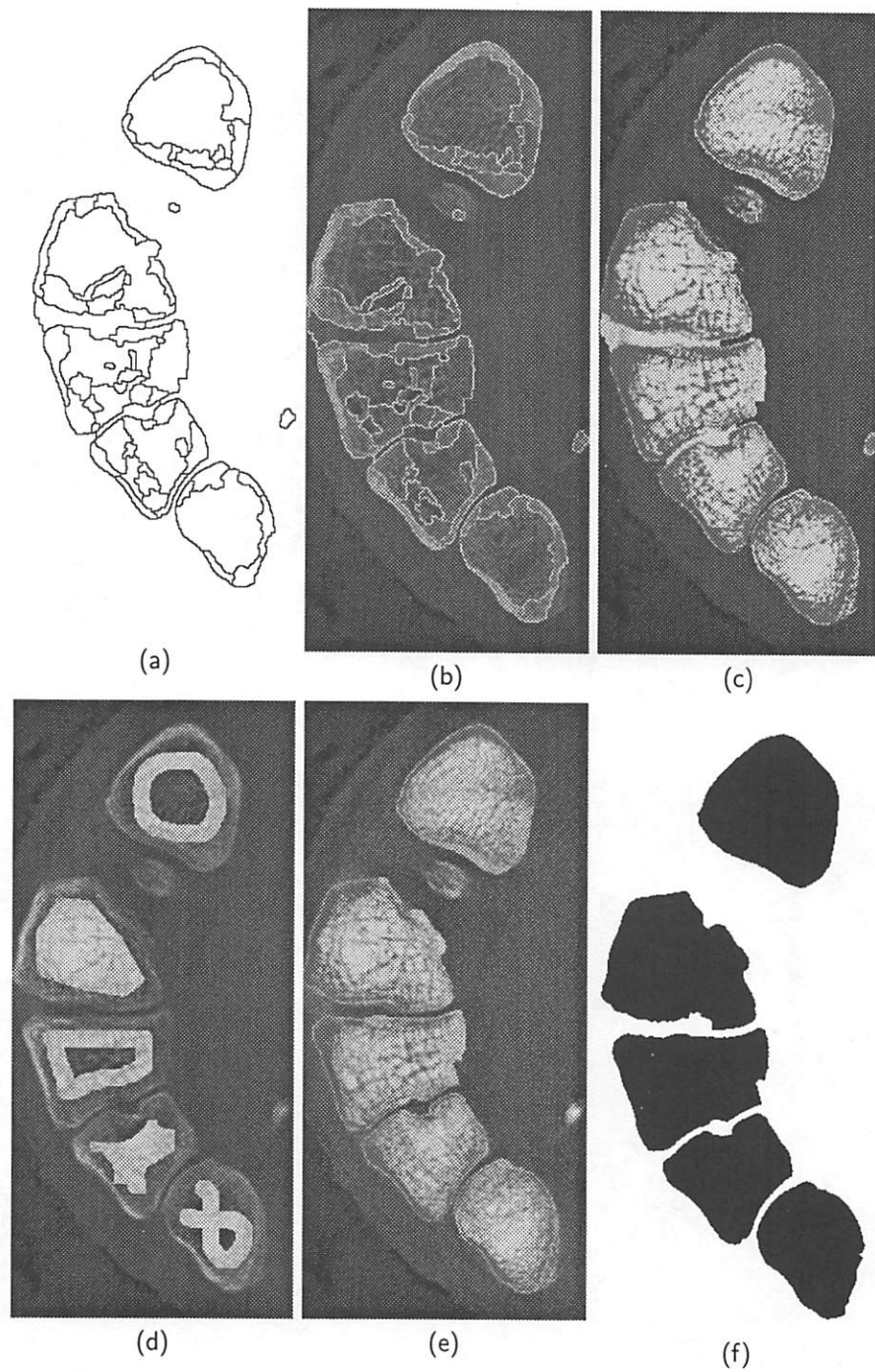


Figure 2: Cross-section of a hand: (a) contours from an off-line segmentation. (b) contours overlaid on the image. (c) Regions created from these contours. (d) Coarse manual initialization of overlays. (e) Interactive segmentation result after region-growing. (h) Resulting mask.

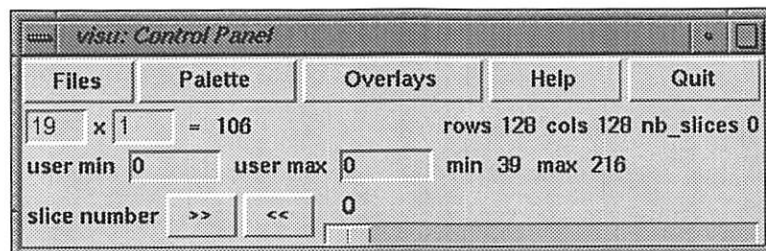


Figure 3: Control Panel

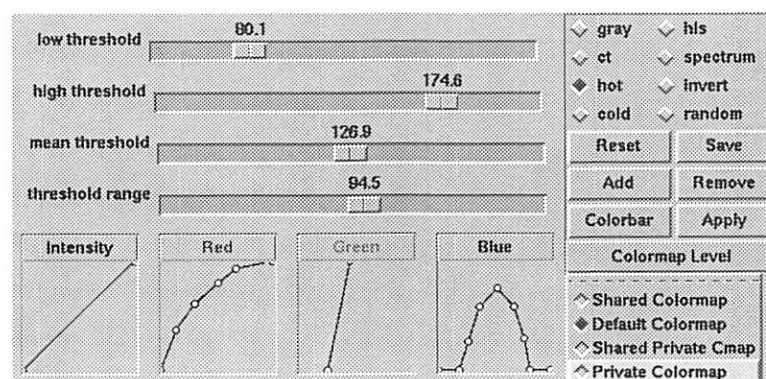


Figure 4: Palette options

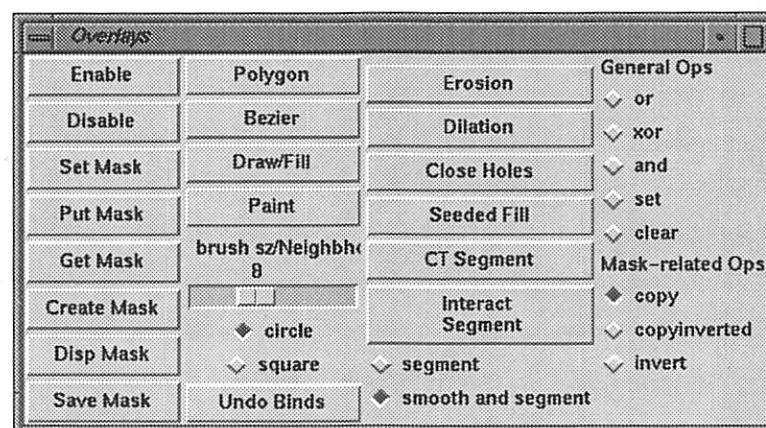


Figure 5: Overlays options

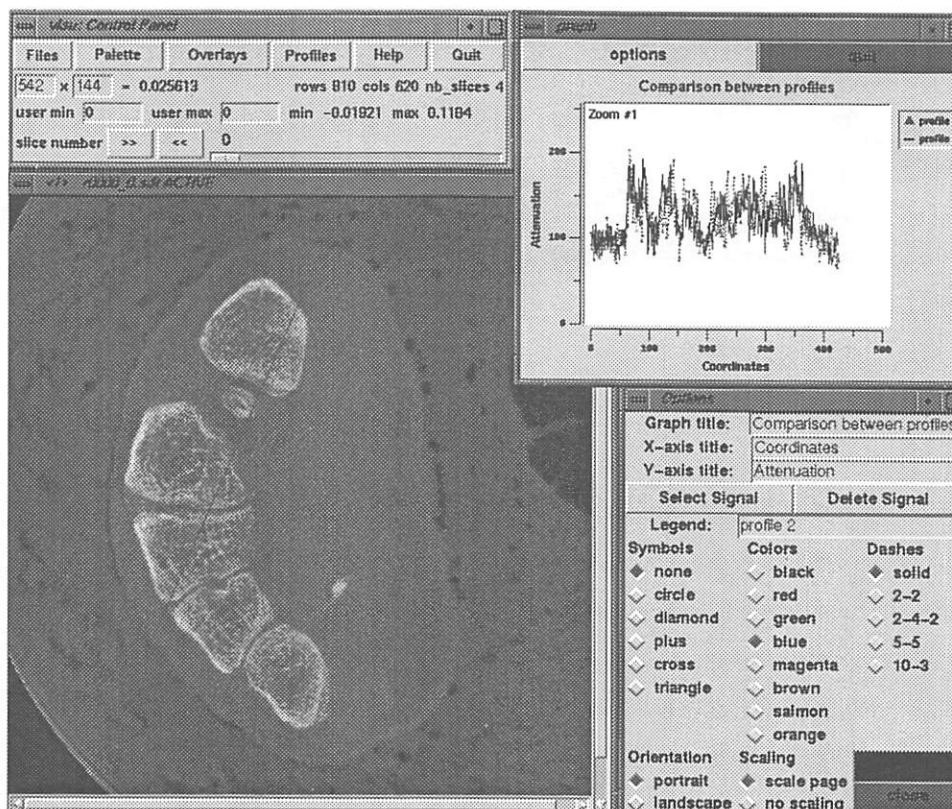


Figure 6: The profiles extracted in VISU are sent across the network to a graph viewer

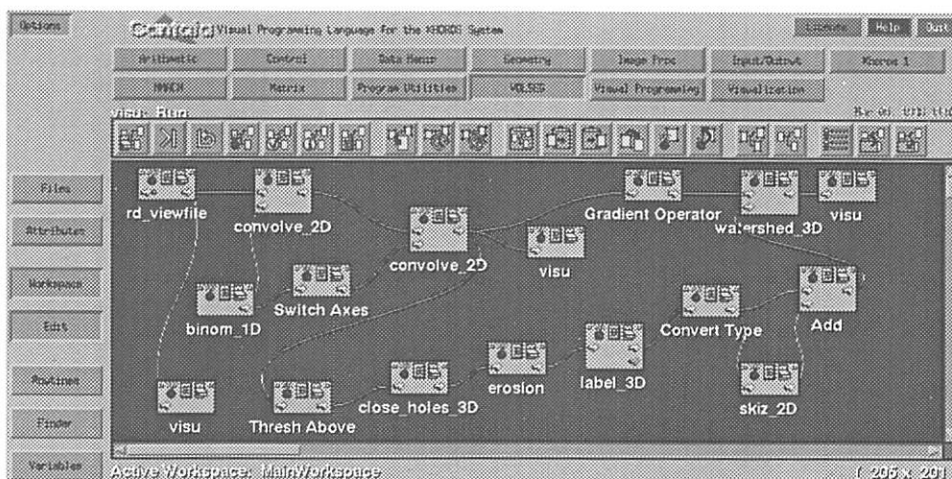


Figure 7: Khoros data flow environment

A Clinical Neurophysiology Information System based on Tcl/Tk

Martin B. Andrews and Richard C. Burgess

*Section of Neurological Computing
Department of Neurology
The Cleveland Clinic Foundation
9500 Euclid Ave
Cleveland, OH 44195
andrewm@ccfadm.eeg.ccf.org*

Abstract

The Department of Neurology at the Cleveland Clinic has replaced a restricted patient information system with a new one based on Tcl/Tk. Tcl/Tk along with several novel extensions provides the base for creating portable client-server database applications. These extensions and some of the clinical applications are presented.

1. Introduction

The Cleveland Clinic is a large multi-specialty medical center with a high volume of both inpatient and outpatient evaluations. One of the largest departments, Neurology, is a leader in the care of patients with epilepsy and other neurological disorders. Along with physical examinations, imaging studies and laboratory tests patients are evaluated with neurophysiological tests like EEG (brain wave tests) and PSG (sleep tests). Tests are performed using computer-based acquisition and review systems developed by the Section of Neurological Computing. This section is responsible for the information needs of the department, and provides networked databases to physicians, nurses, technologists and secretaries. These systems and software products have also been made available for use at other institutions through the Vanguard subsidiary.

The Neurology department's patient information system, EBase, was integral to daily operations and research studies. It was also unreliable, difficult to use and even more difficult to program. A major overhaul of the code was enacted to remove bugs and hardware dependencies. The resulting system, EBase2, was more reliable but development was still painfully slow. Particularly hard to develop were the highly customized user interactions demanded by our users. To overcome this problem a radical redesign of EBase was initiated. Informix-4GL, the proprietary programming language used in earlier revisions, was dumped. The requirements for the new EBase3 were as follows:

- **Rapid Development** - The burgeoning development of new clinical tests, a desire for

faster reporting of test results and a need to reduce costs drive an ever growing demand for information to be maintained by EBase. In addition, vague user requirements necessitate quick prototyping. Programmers must be able to quickly extend EBase.

- **Ease of Use** - Our users demand an intuitive and stream-lined user interface. These somewhat contradictory needs can be met with flexible user interface components that are customized for particular groups of users. Even further flexibility is required if other institutions are to use EBase.
- **Portability** - Departmental users access EBase from Unix workstations, PCs running Microsoft Windows, and Wyse 60 terminals attached to a Unix server. All of these devices must be supported. For EBase to be used at other institutions it must be portable to even more client devices, and even a variety of server platforms.
- **Simple API** - The section of neurocomputing also develops software to acquire and review neurology test data. A simple API for accessing the EBase database is desired for use in these systems.
- **Inexpensive Licensing** - The cost of development tools was an important issue because of our own budget limitations and, more importantly, the need to provide a competitively priced package to other institutions. Even tools without run-time fees

generally charge per target platform - a problem considering the number of targets to be supported.

The resulting design for EBase3 uses Tcl to help meet these goals. The core of EBase3 is an extended Tcl interpreter called Clavier. Clavier controls all access to the patient database. It enforces data integrity rules and hides implementation details of the database schema. Only Clavier sends SQL commands to the database engine. Clients send requests to Clavier via TCP/IP in the form of Tcl commands. The client itself does not require a Tcl interpreter, but the most common client is an extended Tcl interpreter called Bach. Bach is a generic tool for building GUI database applications. Applications for Bach are Tcl scripts. A version of Bach exists for X based workstations, and character terminals connected to a Unix server. A version for Windows based PCs is being developed. Clients that do not include a Tcl Interpreter use a small C library to send Tcl requests to Clavier. The components of EBase3 are depicted in figure 1. The design uses the three-tiered architecture being embraced by packages like PowerBuilder[1].

An elegant feature of the EBase3 design is that the Bach client can load all the Tcl code from Clavier. This feature simplifies maintenance by providing a single point of installation, and allows easy operation over a wide area network.

The remainder of this paper consists of three major sections. Section 2 describes the Tcl extension packages used in the Bach server and client. Section 3 presents some applications implemented using the Bach client. Section 4 concludes with observations from the EBase3 project.

2. Tcl Extensions

Clavier and the Bach client include many Tcl extensions to simplify the creation of an information system. Both Clavier and Bach include the object-oriented programming extension OTcl [2]. The PC and workstation versions of

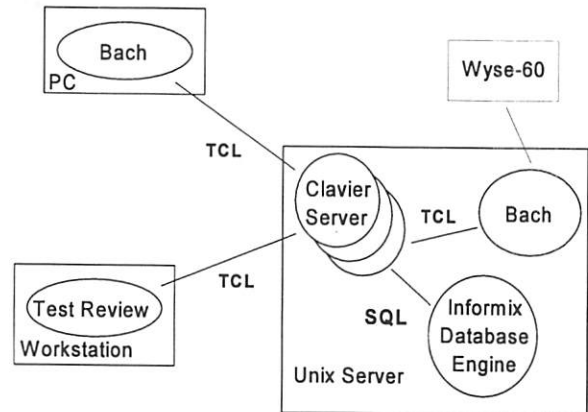


Figure 1. EBase3 Components.

Bach include Tk as the basis for their user interface. The character terminal version of Bach includes CTk, a curses based replacement for Tk that is discussed in the next section. All Bach versions include the table geometry manager from BLT that simplifies the layout of forms, and Stephen Uhler's HTML library for rendering formatted text. In addition to these outside-developed extensions, several new extensions have been developed at the Cleveland Clinic. These in-house extensions are presented in the following subsections.

2.1 CTk

Curses Tk, or CTk, is a replacement for Tk that displays to character terminals. The user interface is navigated using keyboard bindings mostly compatible with Tk (some differences occur because most terminals do not support the range of modifier keys used in Tk). While not as pleasant to use as Tk, CTk provides an important fall back when Tk cannot be used. Besides the obvious case when only a character terminal is available, CTk has also proven useful for remote access. A screen snapshot of the EBase3 schedule application running under CTk is shown in figure

```

schedule
Day Test

```

	8:00	9:30	11:00	1:30	3:00	Night
Prior	Stokes5	-	Brookhouse5	-	White5	Serafin1
Next	Phillips5	Agins5	Rosipko Jr.5	Monjeau5	Nagy5	-
reLoad	-	-	Beckner5	-	Sauerland5	Wimmer1
Go...	-	-	-	-	-	Brown1
Report>	-	-	-	-	-	Douglas1
Exit	-	-	-	-	-	Baxter1
EP-OR	Standard	ght5	-	-	-	-
EP-Rtne	Administrative	-	-	-	-	-
EP-MlIn	Results	ter1	-	Showalter1	-	-
WADA	Units by Tech	ov5	-->	-	-	-
Ambul	Units by Type	-	-	-	-	-
EMU	-	-	-	-	-	-
PMU	Weldy1	Knepp1	Shulman5	Grudny1	Andrews1	-
	White	Hoynes1	Morell1	Sulenski1	-	-

1) Start 2) Tech-Class 3) Phys-Class 5) Final

Figure 2. EBase3 Schedule application using CTk.


```

informix connect myConn "mydatabase"
myConn prepare mySelect {
    select id, first_name, last_name from employees
    where last_name matches ?
    order by last_name, first_name
}
mySelect execute A*
while {[mySelect into vals]} {
    puts "$vals(id) - $vals(last_name), $vals(first_name)"
}

```

Figure 3. Tcl script using the Rdb command "informix". It prints all employees whose last name begins with "A".

2. Like Tk, Ctk is written in C except for event bindings, which are defined in Tcl.

CTk supports the majority of Tk commands, but the differences require existing Tk applications to be modified to run under CTK. It is easy to write applications that will work with both Tk and CTK if this goal is considered from the outset. For an application to be portable to Tk and CTK, the most important requirements are resolution independence and support for small screens. These attributes are useful even for applications solely intended for Tk.

2.2 Rdb

Rdb provides a Tcl interface to relational databases. Currently, Rdb only provides an interface to Informix. Rdb is loosely based on DBI [3], the universal database interface being developed for perl, so it is intended to interface with a variety of database engines. An example Tcl script using Rdb is shown in figure 3. Rdb includes the flexible error handling scheme from DBI that supports recovery from failed database commands. It also supports access to BLOBs. Rdb is implemented entirely in C.

2.3 Bach

The Bach extension is a library of database utilities used in both Clavier and the Bach client. It defines network commands and OTcl classes for domains, fields, forms and reports. Bach is implemented in a mixture of C and Tcl.

The Bach network commands provide a simple method for sending Tcl commands via TCP/IP. In addition, the Bach library includes a short wrapper for the Tcl source command that allows the Bach client to auto-load Tcl files from Clavier.

Domains are OTcl objects that provide easy access to a database. A domain represents a set of member values, and

defines a list of attributes for its members. Each attribute itself has an associated domain. For example, date is a domain and one of its attributes is year. The domain of the year attribute is integer. In addition to algorithmic domains like date and integer, Bach includes the SqlTable class for domains representing the rows of a relational database table. Bach automatically defines parameterized types like character(20), the set of all character strings with length 20 or less, by patching the unknown Tcl command. Domains also specify default fields, forms, and reports for depicting members of the domain.

Fields are widget classes that can take on any value from their assigned domain. All fields provide methods for storing, retrieving, and validating their value. Forms are mega-widget classes containing one or more fields. A simple form layout can be generated automatically, or an intricate layout can be specified explicitly.

The Report class provides the basis for generating HTML formatted reports. Using HTML provides a natural method for browsing records in the database. HTML is used for all formatted text in EBase3, removing the need for separate code for printing or viewing a record. To facilitate printed reports, non-standard mark-up tags and attributes are used to control features like page breaks and page orientation. A custom HTML formatter interprets these non-standard notations and outputs the report in PostScript, HP-PCL, or ASCII format. The HTML report formatter is an independent program implemented in C.

3. Applications

EBase applications are Tcl scripts executed by the Bach client. These applications share a common library of Tcl code that defines domains specific to EBase, and their associated user interface components. The EBase library currently consists of 8000 lines of Tcl code, over half of which is for the complex domain representing neurology tests. Neurology tests have thirty attributes including

complex items like findings and classification that contain many attributes themselves. The actual format of these complex attributes is determined by the study type of the test. The neurology test form is depicted in Figure 4.

The Patient application, shown in Figure 5, provides easy access to all EBase records associated with a single patient. Users are presented with the patient demographics and a concise medical history. Selecting a history entry opens a hypertext browser. At any point, a form can be opened to update or add new patient records. The schedule application, shown in Figure 6, presents neurology tests by the date they were performed. In addition to scheduling or rescheduling tests, the user can enter the hypertext browser by selecting one of the entries. In addition, the schedule application provides a collection of administrative reports. Both of these applications are short Tcl scripts: the patient script contains 221 lines, the schedule script 376 lines.

4. Conclusion

EBase3 has succeeded in creating a portable information system that is easy to use and extend. It is in production use in an environment that schedules 800 tests per month and averages a dozen active users. The new system has been greeted enthusiastically by a wide range of users including secretaries, technologists, physicians, and staff in training.

The tools created for EBase3 have also shown promise for use in other information systems. Clavier and the Bach client were used to create a modest application to generate

Figure 4. Form for editing Neurology Tests.

project status reports. This application manages six database tables, and includes five forms and one report. It required only 550 lines of Tcl code.

I initially considered Tcl for use in EBase3 not because of any attribute of Tcl itself, but because of the Tk Toolkit. While the Tk toolkit has succeeded in providing a rich user interface for EBase3, Tcl itself has proven well suited to

Figure 5. EBase3 patient application.

schedule						
Day Test						
1 Feb 1996						
	8:00	9:30	11:00	1:30	3:00	Night
Lab-1	Spikes5	-	Brookhouse5	-	White5	Serafin1
Lab-2	Phillips5	Agins5	Rosipko Jr.5	Monjeau5	Nagy5	-
Lab-3	-	-	Beckner5	-	Sauerland5	Wimmer1
Portable	-	McKnight5	-	-	-	Brown1
MSLT	Hass Jr.1	-	-	-	-	Douglas1
EEG-OR	-	-	-	-	-	Baxter1
EP-OR	Vincent5	Showalter1	-	Showalter1	-	-
EP-Rtne	-	Mostov5	-->	-	-	-
EP-Mlin	-	-	-	-	-	-
WADA	-	-	-	-	-	-
Ambul	-	-	-	-	-	-
EMU	Weldy1	Knepp1	Shulman5	Grudny1	Andrews1	-
PMU	White	Hoynes1	Morell1	Sulenski1	-	-

1) Start 2) Tech-Class 3) Phys-Class 5) Final

Figure 6. EBase 3 schedule application using Tk.

departmental information systems, where rapid development is usually more important than blazing performance and most of the work consists of string manipulation and interacting with the user. The main problems discovered with Tcl are listed below.

- Performance - While performance was not the main concern in EBase3, there were times where Tcl was too slow even for high level functions. Performance problems have been overcome by recoding commands in C, but this is not an ideal solution. In addition to the greater development effort required by C, C code must be built and installed on all of the client platforms. Tcl code only needs to be installed on the Clavier server. The eagerly awaited on-the-fly Tcl compiler will hopefully solve this problem. Tight integration with Java is another possible solution.
- Development Tools - GUI development tools for Tcl would further the goal of rapid application development, and greatly assist in bringing new programmers up to speed. The SpecTcl project [4] at Sun Labs addresses the need for a GUI builder. A source browser (supporting object-oriented programming) and a debugger are also needed. Production grade tools are a must.

References

[1] Powersoft, *Building Distributed PowerBuilder Applications*, Powersoft, Concord, MA, 1995.

[2] David Wetherall and Christopher J. Lindblad, *Extending Tcl for Dynamic Object-Oriented Programming*, 1995 Tcl/Tk Conference Proceedings.

[3] Tim Bunce, *Perl 5 Database Interface (DBI) API Specification*, Draft Version 0.6, 1994.

[4] Stephen Uhler, *A Graphical User Interface Builder for Tk*, 1995 Tcl/Tk Conference Proceedings.

